Sistematização da Animação de Programas

Proposta de um novo sistema para construção automática e sistemática de animações de programas

Dissertação submetida à Universidade do Minho para obtenção do grau de doutor em Informática, ramo Tecnologia da Programação

Supervisão: Prof. Doutor Pedro Henriques
DEPARTAMENTO DE INFORMÁTICA
UNIVERSIDADE DO MINHO

Maria João Tinoco Varanda Pereira

Dezembro 2002

Resumo

Este documento apresenta e discute a tese de doutoramento da autora. Defendendo a importância da visualização e animação de programas, esta dissertação aborda temas relacionados com a representação visual, estática e dinâmica, dos conceitos envolvidos nos programas de computador: variáveis, operações, instruções de entrada/saída e de controlo, fluxo de dados e de execução. O trabalho teve como principais objectivos aprofundar conhecimentos sobre os conceitos de animação e visualização de programas, rever os sistemas de animação existentes e propor algo inovador nesta área, no sentido da automatização e generalização do processo de construção dessas animações. Por automatização entende-se a capacidade de criar o visualizador/animador a partir de um programa fonte, sem custo adicional para o utilizador. Por generalização entende-se a sua adaptação a diferentes algoritmos e linguagens fonte.

Para além do estudo do estado actual da arte de animar programas, foi criado um sistema de classificação dos animadores existentes e foram efectuadas análises comparativas dos mesmos. É então proposto um novo sistema chamado Alma que separa o processo em *front-end* e *back-end* e usa uma representação intermédia universal para atingir a sistematização pretendida. No documento são apresentadas todas as especificações relativas a este sistema, assim como, diversos detalhes técnicos da sua prototipagem.

A arquitectura concebida para o Alma, baseada em motores e regras de transformação independentes, concede-lhe um caracter extensível sendo, por isso, possível adaptá-lo facilmente a diferentes visualizações e diferentes paradigmas.

Abstract

The PHD Thesis of the author is presented and discussed in this document. In this work, the author defends the usefulness of program animation and visualization, discussing subjects related with the visual representation of the concepts present in computer programs, such as: variables, operations, input and output statements, data and control flow.

After getting a deeper knowledge about animation and visualization of programs, the work proceeded studying existing animation systems, attaining the definition of a classification criteria. Applying that criteria, those systems were grouped by their main characteristics, allowing to compare them and the identification of a class where a solution was lacking. The main contribution of this thesis is a system aiming at filling that gap. This proposal intends to automatize and generalize the construction of animation systems from the formal definition of programming languages. Automatization means the ability to generate a visualizer and animator program for a given programming language from its grammatical specification without the human intervention. Generalization means the ability of the generator to couple with different algorithms and programming paradigms. The proposed visualizer and animator is not algorithm neither language dependent; it processes any program in languages for which an appropriated parser was generated. Moreover, the animation does not require the use of special data types nor special annotations.

The new system, called Alma, separates the *front-end* process from the *back-end* one and uses an universal intermediate representation in order to achieve the desired systematization. In this document, the system specification and technical details about Alma prototype are discussed.

The system architecture, based on independent transformation rules and engines, allows to obtain an open system: it is easy to modify or add rewriting rules and drawings in order to process programs in different paradigms and to produce different visual results.

Agradecimentos

Todos os agradecimentos seriam poucos para reconhecer o constante e competente apoio do meu orientador Prof. Doutor Pedro Henriques na concepção e revisão desta dissertação. Agradeço também todo o apoio financeiro e logístico relacionado com as participações em conferências e simpósios. Sempre que foi possível, fez questão ele próprio de me acompanhar nessas saídas, permitindo assim um trabalho conjunto muito mais intenso, quer pelas horas livres onde aproveitamos para trabalhar, quer pelos contactos que estabelecemos com investigadores de outras instituições. Para além disso, e como dinamizador do grupo de Especificação e Processamento de Linguagens (gEPL) do Departamento de Informática da Universidade do Minho, onde desenvolvi esta dissertação e do qual faço parte, criou um ambiente propício à troca e amadurecimento de ideias, constituindo um bom contributo para o desenvolvimento deste trabalho.

O meu agradecimento vai também para o Prof. Doutor Marjan Mernik e Prof. Mitja Lenic da Universidade de Maribor, Eslovénia, pelo apoio prestado na criação do protótipo desenvolvido no âmbito desta tese.

Reitero o meu reconhecimento aos meus colegas do Departamento de Informática e Comunicações da Escola Superior de Tecnologia e Gestão de Bragança, em especial ao meu colega de gabinete Rufino, por todo o apoio técnico e psicológico, ao Paulo Matos pelo cuidado e compreensão com que elaborou a distribuição de serviço docente, ao Albano, à Dr. Luísa Miranda, ao José Adriano e a todos os colegas em geral, por tornarem a nossa escola um local aprazível e acolhedor onde é bom trabalhar. Agradeço de um modo muito especial à minha família, aos meus pais, ao Rolando, ao Tiago e à Patrícia, e a todos os meus amigos, por serem a base de sustentação que me permite crescer.

Os meus agradecimentos vão ainda para o programa PRODEP, pela bolsa de doutoramento que me concedeu no âmbito da medida 5, acção 5.2 - doutoramentos, e para o Instituto Politécnico de Bragança pelo apoio financeiro nas deslocações ao estrangeiro.

Conteúdo

1 Introdução **19**

1.1 Fundamentos da escolha do tema 20

1.2 A Tese 20

1.3 Identificação dos contributos originais da tese 21

1.4 Divisão da tese por capítulos 21

1.5 Roteiro de leitura 23

2 Estado actual da arte de animar programas **27**

2.1 Animação versus Visualização 27

2.2 Abordagens adoptadas em alguns sistemas de animação 29

2.2.1 O paradigma *Path Transition* 29

2.2.2 Uma abordagem *bottom-up* para visualizar o comportamento de programas . . . 29

2.2.3 Abordagem declarativa de construção de visualizações 30

2.3 Técnicas de construção da animação 30

2.3.1 Uso de bibliotecas de funções de visualização 31

2.3.2 Manipulação directa da animação 31

2.3.3 Anotação de algoritmos 31

2.3.4 Tipos de dados auto-animados 31

2.3.5 Linguagens específicas de animação 32

2.3.6 Anotação da semântica de programas 32

2.4 Classificação dos sistemas de animação de programas 32

2.4.1 Tipo 0 33

2.4.2 Tipo I 33

2.4.3 Tipo II 33

2.4.4 Tipo III . . . 33
2.4.5 Tipo IV . . . 34
2.5 Sistemas de animação . . . 34
2.5.1 Tipo 0 . . . 34
2.5.2 Tipo I . . . 44
2.5.3 Tipo II . . . 48
2.5.4 Tipo III . . . 50
2.5.5 Tipo IV . . . 60
2.6 Animação em Geradores de Compiladores . . . 60
2.6.1 O sistema LRC . . . 60
2.6.2 O sistema CENTAUR . . . 61
2.6.3 O ambiente SmartTools . . . 62
2.6.4 O sistema LISA . . . 63
2.7 Análise dos sistemas de animação do Tipo III . . . 63
2.7.1 Parâmetros de análise . . . 64
2.7.2 Estudo comparativo dos sistemas de animação do tipo III e do sistema proposto . 67
2.8 Conclusão sobre o actual estado da arte . . . 71

3 O sistema Alma **73**
3.1 Apresentação do sistema . . . 73
3.2 Princípio, evolução e objectivos do sistema Alma . . . 75
3.3 O caracter genérico do sistema . . . 76
3.4 Finalidades do Sistema . . . 78
3.5 Características da interface para um sistema deste tipo . . . 79
3.5.1 Características essenciais das interfaces de um SA . . . 79
3.5.2 Interfaces para inserção de programas e escolha de visualizações . . . 81

4 Especificação do sistema Alma **83**
4.1 Arquitectura do sistema . . . 83
4.2 A DAST como representação intermédia . . . 86
4.2.1 Definição formal da árvore de sintaxe abstracta decorada . . . 87
4.2.2 Definição formal da Tabela de Identificadores . . . 88

4.3 Construção da DAST: o *front-end* 89
4.4 Construção da animação: o *back-end* 93
4.5 Visualização no sistema Alma 94
4.5.1 Regras de visualização 94
4.5.2 Algoritmo de construção das visualizações 96
4.6 Animação no sistema Alma 97
4.6.1 Regras de reescrita 97
4.6.2 Algoritmo de reescrita 98
4.7 Gramática Abstracta e Bases de Regras 100
4.7.1 Gramática da representação interna do Alma 100
4.7.2 Regras de reescrita 102
4.7.3 Regras de visualização 105
4.8 Animação de subprogramas 108
4.9 Tratamento de variáveis estruturadas 116
4.10 Exemplos de visualizações possíveis 117

5 Desenvolvimento do Sistema 125

5.1 Estratégia de implementação 125
5.2 Utilização do sistema LISA na geração dos *front-end*'s 126
5.2.1 Escolha do sistema LISA 127
5.2.2 Especificação de linguagens no sistema LISA 128
5.2.3 Extensão da gramática para construção dos *front-end*'s 131
5.3 Reutilização do sistema LISA no BE do sistema Alma 134
5.4 Alguns detalhes de implementação 136
5.4.1 A DAST gerada 136
5.4.2 Algoritmo principal do *back-end* 138
5.4.3 Bases de regras 139
5.4.4 Tabela de Identificadores 141
5.4.5 Visualizador do sistema LISA 142
5.5 O *back-end*: numa perspectiva de Máquinas Virtuais 146

6 Diferentes níveis de utilização do Sistema Alma **151**

6.1 Utilização básica do sistema Alma 152

6.2 Utilização avançada do sistema Alma 152

6.3 Construção de novos FE's 153

6.4 Diferentes tipos de animações/visualizações 154

6.4.1 Diferente nível de detalhe da animação 154

6.4.2 Diferentes tipos de visualizações 156

6.4.3 Diferente tipo de linguagem (paradigma) 161

6.4.4 Como programar o que se quer acrescentar? 171

7 Conclusão **181**

7.1 Objectivos atingidos 181

7.2 Objectivos a alcançar em tempos futuros 182

7.3 Importância pedagógica do sistema Alma 183

A Lista de funções usadas nas especificações do Alma **191**

B Gramática estendida para construção de um *front-end* **193**

C Definição das funções para construção dos nodos da DAST **197**

D Programa principal do sistema Alma **201**

E Exemplos de utilização do sistema Alma **207**

E.1 Primeiro exemplo 207

E.1.1 Código Fonte 207

E.1.2 Visualizações geradas pelo LISA 207

E.1.3 Animação gerada pelo Alma 208

E.2 Segundo exemplo 211

E.2.1 Código Fonte 211

E.2.2 Animação gerada pelo Alma 214

F Extensão da gramática do mini-prolog (construção da DAST) **219**

G Exemplos de utilização do sistema AGG **223**

H Código de animação de um algoritmo Depth First (JAWAA) 227

Lista de Figuras

1.1 Primeira rota de leitura 23
1.2 Segunda rota de leitura 23
1.3 Terceira rota de leitura 24
1.4 Quarta rota de leitura 24
1.5 Quinta rota de leitura 25

2.1 Sistema de Animação 28
2.2 Animação do algoritmo Fibonacci 35
2.3 Programa visual da animação (Handmove) 38
2.4 Primeiro exemplo de um programa AGG 40
2.5 Segundo exemplo de um programa AGG 41
2.6 Algoritmo Bubblesort 46
2.7 Algoritmo Graham's Scan 47
2.8 Fenómeno de Gibbs numa série de Fourier 47
2.9 Animação de grafos 48
2.10 Inserção e remoção de elementos de listas em C 49
2.11 Animação de pilhas 50
2.12 Procura de elementos em grafos 51
2.13 Sistema VIP 54
2.14 Parser LR(1) 59
2.15 Parser LR(1) (continuação) 60

3.1 Concepção do Sistema Alma 77

4.1 Arquitectura do Sistema Alma 84

4.2 Linha de produção da animação a partir do texto fonte 85
4.3 Estrutura dos nodos da DAST 87
4.4 A DAST gerada para o exemplo 4.3.1 91
4.5 A DAST gerada para o exemplo 4.3.2 93
4.6 Visualização de uma operação relacional 96
4.7 Regras de reescrita para instrução condicional 99
4.8 Regras de reescrita (Cálculo de operações) 103
4.9 Regras de reescrita (Cálculo de atribuições) 103
4.10 Regras de visualização (atribuição e entrada/saída) 106
4.11 Regras de visualização (operações) 106
4.12 Representação abstracta de subprogramas 109
4.13 A primeira árvore de execução para o exemplo 4.8.1 110
4.14 Árvore de execução para o exemplo 4.8.1 111
4.15 Regras de reescrita para invocação de subprogramas 112
4.16 Regra de reescrita para o retorno de subprogramas 113
4.17 Árvore de execução com subprogramas calculados 114
4.18 Regras de visualização para subprogramas (nodo RETURN) 115
4.19 Regras de visualização para subprogramas 115
4.20 Visualização de subprogramas 116
4.21 DAST gerada pelo *front-end* para o exemplo 4.10.1 119
4.22 Visualização criada para o exemplo 4.10.1 119
4.23 DAST gerada pelo *front-end* para o exemplo 4.10.2 120
4.24 Alguns passos da animação do exemplo 4.10.2 121
4.25 DAST gerada pelo *front-end* para exemplo 4.10.3 122
4.26 Animação do exemplo 4.10.3 123
4.27 Animação do exemplo 4.10.3 124

5.1 Arquitectura do sistema Alma 126
5.2 Geração do animador a partir de uma especificação LISA 133
5.3 DAST gerada pelo *front-end* para o exemplo 5.2.1 134
5.4 Arquitectura do sistema LISA 134
5.5 Ligação entre o sistema LISA e o sistema Alma 135

5.6 Estrutura dos nodos da DAST 136
5.7 Estrutura dos modelos de nodos que constam nas regras 140
5.8 Estrutura auxiliar das regras 140
5.9 Texto fonte do primeiro exemplo 142
5.10 Imagem inicial da animação do primeiro exemplo 143
5.11 Últimas visualizações do primeiro exemplo 143
5.12 Texto fonte do segundo exemplo 144
5.13 Primeira visualização do segundo exemplo 144
5.14 Visualização da primeira iteração do ciclo 145
5.15 Visualização da segunda iteração do ciclo 145
5.16 Última visualização da animação do segundo exemplo 146
5.17 Processo de construção das visualizações 147

6.1 Última visualização gerada para o exemplo 6.4.1 157
6.2 Uma visualização diferente para a primeira instrução do exemplo 6.4.1 157
6.3 Uma visualização diferente para a segunda instrução do exemplo 6.4.1 158
6.4 Nova visualização do exemplo 6.4.1 usando novas primitivas de desenho 158
6.5 A DAST gerada para o exemplo do robot 160
6.6 A tabela de identificadores para o exemplo do robot 160
6.7 As várias imagens da animação do robot 161
6.8 Árvore gerada pelo sistema LISA para o exemplo 6.4.3 164
6.9 Visualização criada para o exemplo 6.4.3 164
6.10 Visualização clássica para o exemplo 6.4.4 165
6.11 Nova visualização para o exemplo 6.4.4 165
6.12 Árvore de execução para o exemplo 6.4.5 167
6.13 Nova árvore de execução para o exemplo 6.4.5 167
6.14 Animação criada pelo sistema ALMA 168
6.15 Regras de reescrita para unificação de predicados e propagação de valores 169
6.16 A DAST gerada para o programa em Haskell 171
6.17 Resultado final da animação do programa em haskell 171

E.1 Visualização do autómato da análise léxica 208

E.2 Visualização de parte da gramática concreta 209
E.3 Visualização de parte da árvore de sintaxe 209
E.4 Visão estruturada do programa fonte 210
E.5 Visualização da DAST gerada pelo *front-end* do Alma 210
E.6 Primeira visualização da animação do primeiro exemplo 211
E.7 Outra visualização da animação do primeiro exemplo 212
E.8 Outra visualização da animação do primeiro exemplo 212
E.9 Outra visualização da animação do primeiro exemplo 213
E.10 Outra visualização da animação do primeiro exemplo 213
E.11 Última visualização da animação do primeiro exemplo 214
E.12 Primeira visualização da animação do segundo exemplo 215
E.13 Outra visualização da animação do segundo exemplo 215
E.14 Outra visualização da animação do segundo exemplo 216
E.15 Outra visualização da animação do segundo exemplo 216
E.16 Outra visualização da animação do segundo exemplo 216
E.17 Última visualização da animação do segundo exemplo 217

G.1 Exemplo AGG com relacionamento de variáveis 224
G.2 Aplicação de regras no exemplo da figura G.1 224
G.3 Segundo exemplo AGG com relacionamento de variáveis 225
G.4 Aplicação de regras no exemplo da figura G.3 225

Lista de Tabelas

2.1 Critérios de Classificação 33
2.2 Classificação de alguns sistemas de animação quanto à área de abrangência 67
2.3 Classificação de alguns sistemas de animação quanto ao conteúdo das visualizações . . . 67
2.4 Classificação de alguns sistemas de animação quanto à forma das visualizações 69
2.5 Classificação de alguns sistemas de animação quanto ao método de especificação das visualizações (Estilo de Especificação) 69
2.6 Classificação de alguns sistemas de animação quanto ao método de especificação das visualizações (Técnicas de Conexão) 70
2.7 Classificação de alguns sistemas de animação quanto à interacção do sistema 70
2.8 Classificação de alguns sistemas de animação quanto a situações de aplicação 71

5.1 Códigos e números de produção dos símbolos da gramática abstracta do Alma 137
5.2 Exemplo de uma tabela de identificadores 141

6.1 Modificações no sistema Alma 152
6.2 Mapeamento entre conceitos da gramática concreta e símbolos da gramática abstracta do Alma 163

Capítulo 1

Introdução

> Podemos até falar só uma língua mas todos dominamos várias linguagens. E quando a linha do caracter se liberta dos códigos miúdos e assume sérios ângulos e curvas insinuantes, aí sim: desorbitamos os olhos e torrentes de ideias (sentimentos, sons, texturas, cheiros e paladares) nos invadem. Porque se a escrita convida, com recato, à viagem, o desenho está na rua, é democrático, público e directo. Todos sabemos onde fica aquele livro que está esquecido. Todos lembramos o desenho desfocado e projectado entre a parede e o tecto na sala de aula. Rasgo depurado do mestre, de quem já fez a viagem e agora revela em cartaz publicitário a rota aventura e principais monumentos. Quem nunca espetou o dedo no mapa de uma cidade desconhecida e disse: " estamos aqui e vamos para ali". O esquema do mais complexo metropolitano do mundo é um icon da arte do século XX. Arte que guia milhões, transporta milhões. Afinal, falamos de sentidos, humanos e físicos, sentidos de conhecimento e racionalidade, mapas do tesouro de X vermelho no complexo labiríntico da sabedoria.

O tema desta tese de doutoramento é *Sistematização da Animação de Programas*. Aceitando como ponto de partida a importância da animação de programas a nível pedagógico, e o interesse que resulta de uma nova representação da informação —a representação visual— este trabalho consistiu na caracterização desses domínios (animação e representação visual) e na exploração de técnicas clássicas de compilação para encontrar uma nova abordagem ao problema em causa. Considera-se que *Animação de Programas* consiste na apresentação visual e textual do controlo de fluxo e dos diferentes valores das variáveis ao longo de uma simulação da execução do programa em análise.

Pretende-se caracterizar detalhadamente este domínio de modo a poder-se estudar a viabilidade de um ambiente genérico para resolução sistemática deste tipo de problemas. Os sistemas de animação de programas existentes foram criados especificamente para determinadas linguagens ou estruturas de dados.

Sendo assim, propõe-se a criação de um sistema para gerar automaticamente animações de programas, cuja arquitectura permite a definição das linguagens que se pretende animar.

1.1 Fundamentos da escolha do tema

A animação de programas (visualização de simulações da execução) poderá ser uma componente essencial no ensino de disciplinas relacionadas com a programação, o que constitui um primeiro motivo de entusiasmo pelo tema. De facto, a animação visual de um programa é útil pelo menos a três níveis: ajuda a compreender a semântica operacional da linguagem fonte; é um auxiliar à depuração de erros (*debugger* de alto nível); facilita o entendimento dos algoritmos. Contudo, o interesse deste trabalho não se baseia só no aspecto pedagógico, mas também em aspectos tecnológicos necessários para alargar estas facilidades a um maior número de linguagens, o que, indubitavelmente, constitui o maior motivo de interesse.

O estudo de uma solução inovadora baseada na especificação formal de linguagens, constitui um novo desafio dentro de uma área onde o grupo onde esta investigação se realizou já tem alguma experiência. A criação de uma arquitectura que permita a generalidade proposta para o novo sistema baseia-se em conceitos subjacentes à área de compilação onde também já existem, no grupo referido, ideias bem consolidadas.

1.2 A Tese

Com o desenvolvimento desta dissertação pretende-se provar a seguinte hipótese: *É possível sistematizar a animação de programas, independentemente do algoritmo subjacente ou da linguagem fonte, permitindo a sua automatização*. Esta hipótese será provada com a concepção de um novo sistema, que concretiza a automatização e generalização do processo de construção das animações. Este processo será automático na medida em que o utilizador apenas indica o programa fonte a animar.[1]. A generalização pretendida é conseguida pelo novo sistema na medida em que é fácil a sua adaptação a novos programas fonte e à produção de diferentes visualizações.

O sistema proposto chama-se Alma e atinge os objectivos apresentados à custa de uma arquitectura que separa o processo em *front-end* e *back-end* e usa uma representação intermédia universal. Esta arquitectura permite ainda atribuir um caracter extensível ao sistema, para obter diferentes visualizações

[1] não precisa de incluir no código fonte nenhum tipo de indicações sobre como ou o que animar

ou suportar diferentes paradigmas.

1.3 Identificação dos contributos originais da tese

Os contributos originais desta tese estão relacionados com a concepção de um novo sistema de animação que, embora tenha objectivos comuns a muitos sistemas já existentes, tenta atingi-los de forma inovadora, expedita e genérica.

O novo sistema de animação pretende ser mais genérico do que os restantes em relação aos algoritmos que suporta e às linguagens fonte que recebe e mais expedito na medida em que não requer qualquer anotação ao código fonte original para a criação de descrições visuais animadas. O sistema poderá ser preparado para receber qualquer tipo de programa e poderá incluir diversos tipos de visualização para os mesmos programas. Argumenta-se, portanto, que o sistema Alma é original, em termos de desenho/objectivos, arquitectura e estratégia de implementação. Além disso, considera-se também original o principio que assegura as características acima referidas: o sistema, em vez de ser como é habitual, orientado para um algoritmo ou uma linguagem, assenta completamente numa representação semântica. Diz-se, por isso, que é orientado ao significado, ou seja, à alma do programa.

Um outro contributo original desta tese consiste na catalogação dos sistemas de animação encontrados e na criação de um sistema de classificação que os caracteriza por tipos.

1.4 Divisão da tese por capítulos

O trabalho subjacente à tese foi essencialmente dividido em cinco grandes etapas. Estas etapas correspondem a cinco tarefas: identificar sistemas de animação existentes e trabalhos efectuados nesta área, criando um novo sistema de classificação que permite caracterizá-los por tipos; conceber um novo sistema de animação e sua arquitectura; especificar todas as componentes do sistema; planear uma estratégia de implementação, identificando as várias fases de desenvolvimento; por último, apresentar resultados concretos da utilização do sistema criado, discutindo extensões à sua implementação para geração de outros tipos de visualizações. A cada uma destas tarefas corresponde um ou dois capítulos: para além da introdução, a tese é composta por mais seis capítulos sendo o último a conclusão.

O segundo capítulo faz uma breve introdução aos conceitos básicos sobre visualização e animação de programas, e apresenta algumas abordagens e técnicas de implementação de sistemas de animação. Sendo o objectivo principal deste capítulo a apresentação do estado actual da arte de animar programas, nele se

reporta a grande variedade de animações e sistemas de animação encontrados ao longo do estudo subjacente. A disparidade de estilos e objectivos identificados nesses sistemas, levantou a necessidade de criar uma grelha de classificação de sistemas de animação que os divida por tipos. Assim sendo, é feita uma relação de sistemas pertencentes a cada tipo. Ainda neste capítulo, são apresentados, de forma breve, alguns geradores de compiladores que produzem algumas facilidades de visualização relacionadas com o compilador gerado. Por último, é feita uma análise comparativa de sistemas usando uma grelha de classificação descrita em [SDBP97] por Stasko.

O terceiro capítulo apresenta o sistema proposto, explica como surgiu este desafio, indica as suas características e finalidades, e identifica os requisitos da interface de um sistema deste tipo.

O capítulo quatro apresenta: a arquitectura do sistema Alma; a especificação formal (gramática abstracta) da representação intermédia (DAST) e a especificação da estrutura que a implementa; o processo de construção do *front-end*; o processo de construção da animação (*back-end*); e a especificação das regras de reescrita e de visualização. Neste capítulo, é ainda especificada a semântica associada a alguns nodos da DAST, assim como, são feitas algumas considerações sobre os nodos representativos de variáveis estruturadas. O capítulo finaliza com a apresentação de alguns exemplos de visualizações que o sistema deve gerar para vários tipos de linguagens fonte.

O capítulo cinco aborda algumas questões relacionadas com o desenvolvimento do primeiro protótipo do sistema, nomeadamente: a estratégia de implementação; a implementação de *front-end* usando o sistema LISA; a reutilização do mesmo sistema para construir o *back-end* do Alma; e outros detalhes da sua implementação - algoritmo principal do *back-end*; programação de regras de visualização e de reescrita; e criação da tabela de identificadores. São mostrados alguns exemplos de visualizações geradas pelo sistema Alma, que também, servem de pretexto para mostrar o visualizador do sistema LISA. Por fim, a arquitectura do Alma é revisitada, agora numa perspectiva de máquinas virtuais e linguagens de domínio específico.

A arquitectura do sistema Alma permite que este seja visto como um sistema aberto a expansões / adaptações, dependendo do nível de utilização: ao nível do utilizador final, o sistema não sofre qualquer alteração, sendo receptivo aos diferentes programas a animar; ao nível de um utilizador intermédio, o sistema permite a construção de *front-ends* para novas linguagens; ou, ao nível de um utilizador mais sofisticado, permite uma intervenção mais complexa, podendo ser preparado para suportar um novo paradigma (criação de novas regras de reescrita e de visualização). No capítulo seis são, então, exploradas essas possibilidades e são dadas algumas indicações de como podem ser feitas modificações no

sistema.

O último capítulo contem a conclusão da tese revendo os objectivos atingidos e identificando trabalho futuro.

1.5 Roteiro de leitura

O leitor desta dissertação poderá guiar a sua leitura consoante os seus interesses e, para tal, terá ao seu dispor os seguintes percursos:

- Para obter algumas noções básicas sobre sistemas de animação e sobre abordagens e técnicas mais vulgarmente usadas na especificação e implementação desses sistemas, sugere-se a leitura da secção 2.1, 2.2 e 2.3 (Figura 1.1).

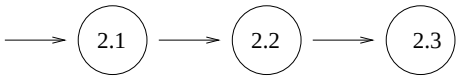

Figura 1.1: Primeira rota de leitura

- Para obter um resumo dos sistemas de animação existentes, sugere-se a leitura da secção 2.5, podendo encontrar alguns pormenores nos apêndices G e H. Os sistemas encontram-se divididos por tipos como resultado de uma classificação apresentada na secção 2.4. Um estudo comparativo dos sistemas recolhidos é apresentado na secção 2.7 (Figura 1.2).

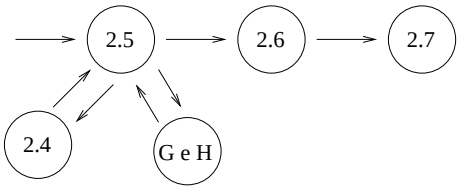

Figura 1.2: Segunda rota de leitura

- Para obter uma breve apresentação do novo sistema, deverá ler o capítulo 3. Deverá prosseguir a leitura pelos capítulos 4 e 5 para compreender todo o processo de especificação e implementação. No apêndice A, onde é apresentada uma lista das funções (e respectivas assinaturas) usadas nas especificações, encontra um suporte teórico ao capítulo 4. No apêndice B é apresentado o código

relativo à construção do *front-end* da linguagem usada nos exemplos do capítulo 5; no apêndice E consta mais um desses exemplos, em relação ao qual se mostram todas as visualizações criadas pelo sistema LISA e a animação gerada pelo sistema Alma; nos apêndices C e D constam os principais ficheiros de código Java do sistema Alma; pelo que estes quatro apêndices devem ser visitados como complemento à leitura do capítulo 5. No capítulo 6 são discutidas questões relacionadas ainda com o sistema Alma e sua extensão, mas sob o ponto de vista do utilizador, pelo que este capítulo pode ser lido depois do capítulo 4 ou 5. Um outro exemplo de construção de um *front-end* (suportando e ilustrando as afirmações do capitulo 6) é dado no apêndice F (Figura 1.3).

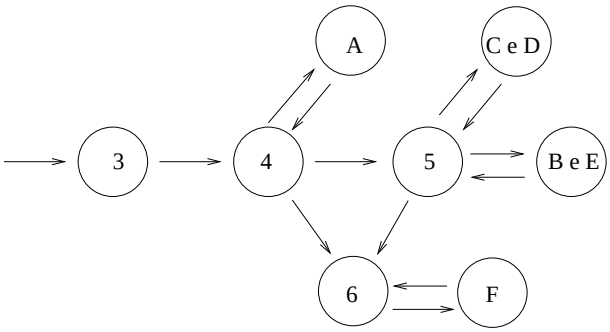

Figura 1.3: Terceira rota de leitura

- Para observar os resultados produzidos pelo sistema para programas exemplo que lhe foram submetidos, poderá debruçar-se apenas na secção 4.10 ou no apêndice E (Figura 1.4).

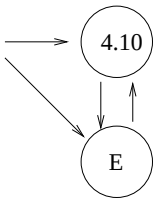

Figura 1.4: Quarta rota de leitura

- Para analisar uma outra visão da arquitectura do sistema Alma, é apresentada, na última secção do capítulo 5 (5.5), uma nova perspectiva baseada em máquinas virtuais (Figura 1.5).

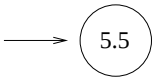

Figura 1.5: Quinta rota de leitura

Capítulo 2

Estado actual da arte de animar programas

Este capítulo pretende fazer uma resenha o mais completa possível: sobre abordagens e técnicas usadas na especificação e implementação de sistemas de animação; sobre os vários sistemas de animação existentes; e sobre geradores de compiladores que também produzem alguma animação. É também apresentado um sistema de classificação, onde se enquadram também as recolhas efectuadas e um estudo comparativo dos sistemas encontrados.

2.1 Animação versus Visualização

Definição 2.1.1 (Visualização de Programas) *Representação gráfica da estrutura (controlo + dados) dum programa, mostrando as características estáticas*

É necessário ver a *animação* como um processo dinâmico e complexo, caracterizado por uma dimensão temporal, com misturas de mudanças contínuas e mudanças ocasionais que produzem uma evolução concorrente de vários objectos gráficos.

Definição 2.1.2 (Animação de programas) *Visualização de estados sucessivos dum programa, representando as cracterísticas dinâmicas.*

Por outras palavras, a *animação de um programa* é um tipo de visualização dinâmica das principais abstracções expressas pelo algoritmo subjacente ao programa; a animação é uma forma natural de representar comportamentos.

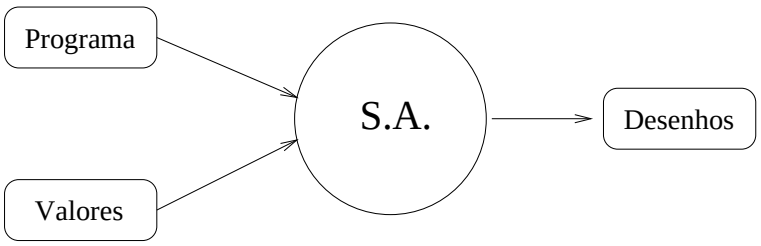

Figura 2.1: Sistema de Animação

A importância da animação de algoritmos reside na habilidade de retratar a essência da lógica do programa. Um dos papeis fundamentais das linguagens visuais é facilitar a percepção e a exploração de informação complexa. Os professores usam, muitas vezes, o poder expressivo das representações visuais para ajudar os alunos a entenderem os algoritmos e o comportamento dos programas.
Uma *visualização* corresponde a uma imagem da animação, a um estado do programa, daí defender que uma *animação* é composta por um conjunto de visualizações.
Para o aluno, a animação do seu programa facilita a sua tarefa de programação, no entanto, a tarefa de programar a animação é ainda árdua e de difícil aprendizagem para a maior parte dos deles. Daí surgirem os sistemas de animação.

Definição 2.1.3 (Sistema de Animação (Figura 2.1)) *Sistema que permite construir animações de forma mais ou menos interactiva com o objectivo de reproduzir comportamentos.*

Um sistema de animação de programas deverá permitir a entrada de programas (textuais) e de valores para os seus dados genéricos e deverá produzir como resultado uma simulação visual ou textual do seu comportamento durante uma execução normal, em que esses dados concretos vão sendo alterados.
Existem vários conceitos relacionados com os sistemas de animação, tal como, programação visual, visualização de programas e animação de algoritmos. A visualização de programas é uma visualização mais baixo nível e a visualização de algoritmos pode ser considerada mais alto nível. A visualização de software (VS) tem como objectivo tornar os algoritmos e os programas mais fáceis de entender. A programação visual é usada na escrita de programas com o intuito de tornar mais fácil essa mesma especificação.
A visualização de algoritmos consiste na visualização de abstracções que descrevem software. Por outro lado, a animação de algoritmos consiste numa visualização dinâmica dessas abstracções de modo a

mostrar a sua evolução ao longo do tempo.

No âmbito da visualização de software pode-se distinguir três personagens que desempenham papeis importantes: o programador da ferramenta de VS; a pessoa que programa a visualização / animação, ou seja, especifica, usando o sistema de VS, de que forma a visualização é aplicada ao programa; e, por último, o utilizador que vê uma visualização estática ou interactua com uma visualização dinâmica do programa.

A nível científico, este não deixa de ser um assunto com interesse e, embora já tenham surgido trabalhos neste âmbito desde 1981, ainda pouco foi feito no sentido de reduzir o caracter específico deste tipo de sistemas. Na verdade, relativamente à animações de algoritmos/programas textuais, todos os sistemas existentes de que se tem conhecimento foram desenvolvidos tendo em vista linguagens específicas e, em certos casos, até só suportam uma determinada família de programas.

2.2 Abordagens adoptadas em alguns sistemas de animação

Nesta secção apresentam-se algumas abordagens relevantes e originais que certos sistemas de animação utilizam na construção das visualizações / animações. É de notar que um sistema concreto pode utilizar mais do que uma destas abordagens simultaneamente, ou não utilizar nenhuma delas.

2.2.1 O paradigma *Path Transition*

Alguns dos primeiros sistemas de animação criados baseiam-se no paradigma *Path Transition*, independentemente da forma como a animação é programada. Este paradigma baseia-se na noção de que uma animação é criada através de um conjunto de alterações feitas sobre uma imagem. Assim, para especificar uma animação será necessário indicar as modificações em cada transição de imagem para imagem. Este modelo foi implementado no sistema Tango criado por Stasko [Sta90] e também serviu de base para a criação da linguagem visual Forms3 [CBC96].

2.2.2 Uma abordagem *bottom-up* para visualizar o comportamento de programas

Esta abordagem surge da necessidade de criar sistemas de visualização com aplicação prática. Os sistemas tradicionais, segundo Hideki Koike e Manabu Aida em [KA95], preocupavam-se com a construção de figuras concretas para domínios de aplicação muito específicos. Os autores propõem uma forma inovadora de construir figuras: em vez do processo *top-down* tradicional que percorre a árvore de sintaxe do

programa e que tem como resultado uma figura final concreta de representação desse programa, usa-se um processo *bottom-up* para que a figura final seja uma conjugação de figuras locais coleccionadas durante esse processo.
Os autores acreditam que um sistema de visualização deve ser o mais genérico possível para ter alguma aplicação prática na programação, ou seja, não deve pôr restrições ao programa escrito pelo utilizador.
O sistema criado por estes autores é um meta-interpretador ao qual são adicionados procedimentos de visualização. Os programas a visualizar não são alterados. À medida que o programa é executado, o sistema constrói figuras locais usando regras de desenho definidas em pontos específicos que tipicamente representam o fluxo de controlo do programa. A figura final é dada por um conjunto de figuras locais. No entanto, poderão surgir situações em que o tamanho da figura ultrapassa o tamanho do ecrã ou se sobrepõe a figuras já existentes. Para resolver tais situações o sistema criado usa um mecanismo de escalonamento.

2.2.3 Abordagem declarativa de construção de visualizações

A chamada *Visualização Declarativa* é uma técnica que permite que o animador construa representações visuais complexas da execução dos programas, definindo mappings entre estados de programas e objectos gráficos.
O programador pode desenvolver visualizações sem examinar o algoritmo porque basta o conhecimento da representação do estado do programa. O papel do *animador* é estabelecer os *mappings* entre estados do programa e objectos gráficos. A separação entre código e visualização traz grandes vantagens em relação a outras abordagens anteriores (como a da anotação do código fonte com primitivas de animação). O processo de visualização é independente das alterações que o código fonte possa sofrer. Alguns dos sistemas que tiram partido desta abordagem são: Pavane [RCWP92], Provide ([Moh88] citado em [SDBP97]), Animus ([Dui98] citado em [SDBP97]) e Aladdin ([HHR89] citado em [SDBP97]).

2.3 Técnicas de construção da animação

Durante o estudo efectuado sobre os sistemas de animação construídos até ao momento, foram surgindo alguns artigos que focam várias abordagens adoptadas por esses sistemas na construção das animações. Nesta secção apresentamos algumas dessas abordagens.

2.3.1 Uso de bibliotecas de funções de visualização

Os primeiros sistemas recorriam a bibliotecas para construir as visualizações de forma não automatizada e pouco sistemática. O utilizador criava ele próprio o código necessário para gerar a visualização com a ajuda de funções pré-definidas. A visualização é gerada em tempo de execução por uma unidade de código que examina o estado dos dados e faz o display da visualização apropriada.

2.3.2 Manipulação directa da animação

O desenhador manipula directamente a apresentação da visualização. Isto involve uma versão limitada de programação visual de objectos gráficos onde o ambiente de interacção reflecte imediatamente a visualização resultante. O sistema LENS ([MS93] citado em [SDBP97]) é um exemplo de um sistema que usa esta técnica. Em geral, todos os sistemas de programação visual que permitam a observação do comportamento desses programas visuais estão, no fundo, a gerar animações de programas.

2.3.3 Anotação de algoritmos

Os primeiros sistemas de animação tinham como base uma anotação exaustiva do algoritmo a ser visualizado, usando primitivas de animação pré-definidas. O programador teria que ter conhecimento sobre quais os procedimentos a utilizar, quais os pontos estratégicos a inspeccionar e a animação era, então, programada directamente no código fonte. O sistema BALSA ([BS84] citado em [SDBP97]) constitui um bom exemplo da utilização desta técnica.

2.3.4 Tipos de dados auto-animados

Numa tentativa de evitar a programação exaustiva de animações, surge uma abordagem no âmbito do sistema JELIOT [HPS+97] que defende o uso de tipos de dados especiais, aos quais chamam *auto-animados*. A programação da animação não recorre a chamadas a procedimentos de animação mas a operações associadas a tipos de dados especiais.

No caso do sistema JELIOT, o código Java (entrada do sistema) fica sujeito a uma pré-compilação que extrai toda a informação necessária e substitui as operações por chamadas a procedimentos de animação. Depois da pré-compilação o código Java é compilado normalmente.

2.3.5 Linguagens específicas de animação

Alguns sistemas de animação, como é o caso do sistema VIP [MM88], obrigam a que o programa a animar seja escrito numa linguagem própria do sistema (pseudo-pascal, pseudo-C,...). Nestas situações não é necessário acrescentar instruções de animação porque o sistema reconhece a sintaxe e a semântica do programa e despoleta automaticamente o processo de construção da animação.

2.3.6 Anotação da semântica de programas

Existe uma abordagem adoptada no âmbito do projecto CENTAUR [Ber91] cujo objectivo é formalizar a semântica de programas, de forma a ser possível associar primitivas de animação e visualizar a execução desses programas. Esta abordagem surge de uma tentativa de evitar uma programação exaustiva de animações, defendendo a associação de primitivas de animação à descrição formal da semântica e não ao programa a visualizar. A *anotação da semântica* surge em oposto à *anotação sintáctica de algoritmos*.

2.4 Classificação dos sistemas de animação de programas

Sendo o objectivo desta tese a criação de um ambiente que sistematize/generalize a animação, tornou-se obrigatório identificar tudo o que tem sido feito a este nível. Dada a diversidade de trabalhos existentes sobre este assunto, foi necessário catalogar abordagens, aplicações e sistemas. Esta catalogação foi feita com base numa classificação por tipos.

O universo que esta classificação pretende abranger é constituído por sistemas relacionados com a visualização e a construção de animações. Foram classificados, neste universo, sistemas de animação específicos, sistemas que permitem ao próprio utilizador a construção das visualizações e, por último, sistemas que criam animações a partir de um texto fonte de forma dependente ou independente da linguagem utilizada.

Os critérios utilizados (ver tabela 2.1) baseiam-se no objecto de animação (genérico ou programas), na possibilidade de interagir com o sistema, na possibilidade de animar diferentes algoritmos e na possibilidade de suportar diferentes linguagens ou paradigmas.

Embora não usados nesta tabela classificativa foram ainda considerados (no estudo de cada tipo) os seguintes parâmetros: se o código fonte é ou não modificado com a introdução de instruções ou tipos de dados especiais; se é feita a geração automática de animações; se é possível indicar parâmetros de animação.

	OBJ.ANIMAÇÃO	INTERACÇÃO	ALGORITMO	LING. DE PROG.	PARADIGMA
Tipo 0	Rep. visuais	Sim	Não se aplica	Não se aplica	Não se aplica
Tipo I	Programas	Não	Específico	Específica	1
Tipo II	Programas	Sim	Específico	Específica	1
Tipo III	Programas	Sim	Genérico	1 (Específica / Standard)	1
Tipo IV	Programas	Sim	Genérico	Várias	Vários

Tabela 2.1: Critérios de Classificação

2.4.1 Tipo 0

Os sistemas aqui representados são do tipo *faça voc̃ê mesmo!* Este tipo representa os sistemas que, embora não tendo sido construídos para gerar animações a partir de programas, permitem a construção das próprias visualizações e animações. Estes sistemas genéricos poderão ser usados na implementação de sistemas de animação de programas, na fase final da construção dos objectos gráficos e sua animação.

2.4.2 Tipo I

Este tipo representa as animações criadas para determinado algoritmo que não permitem qualquer tipo de interacção com o utilizador. Estas animações pré-definidas podem ser usadas em apresentações de programas para explicar o funcionamento de algoritmos ou como sistema de ajuda para quem tenta compreender esses mesmos algoritmos. O utilizador, neste caso, é um mero espectador mas pode, em algumas circunstâncias, dar início, interromper e controlar o andamento da animação.

2.4.3 Tipo II

Este tipo surge em aplicações cuja animação vai sendo progressivamente despoletada como resposta a acções do utilizador, tornando a interacção mais realista. São, por exemplo, animações que permitem inserir valores quando o algoritmo contem instruções de entrada ou, até mesmo, usar o mesmo tipo de animação para comparar diferentes algoritmos para resolução do mesmo problema (por exemplo, diferentes algoritmos de ordenação).

2.4.4 Tipo III

Este tipo diz respeito a sistemas de construção de animações limitados a uma determinada linguagem de entrada. Nestes sistemas podem surgir duas situações distintas: o utilizador usa uma das lingua-

gens tradicionalmente conhecidas (C, Pascal, etc) ou usa apenas a linguagem do sistema, própria para descrever o algoritmo que pretende animar/visualizar. Na primeira situação, o programa a ser animado é, normalmente, anotado com chamadas a procedimentos de desenho. Na segunda situação, o código fonte não é alterado porque a própria linguagem em que o algoritmo é escrito, despoleta a construção da animação.

A animação é gerada automaticamente a partir do texto fonte e não pode ser programada (no caso dos sistemas estudados). Estes sistemas são multi-algoritmo mas não multi-linguagem.

Englobou-se também no tipo III, os ambientes de programação como PECAN ([Rei85] citado em [SDBP97]), GARDEN ([Rei87] citado em [SDBP97]) e FIELD ([Rei90] citado em [SDBP97]) que permitem algumas visualizações de informação relacionada com os programas e que serão abordados mais adiante.

2.4.5 Tipo IV

Este tipo representa os sistemas genéricos multi-linguagem / multi-paradigma. Os programas a serem animados poderão estar escritos em várias linguagens, sendo o sistema capaz de extrair as principais características desses programas e gerar as animações mais adequadas.

Estes sistemas não implicam a alteração do código fonte e a animação é gerada automaticamente.

2.5 Sistemas de animação

Por ser interessante e por poder constituir uma boa base de estudo, vão ser apresentadas nesta secção algumas animações / sistemas de animação pertencentes aos diversos tipos apresentados na secção anterior. É de notar que, o sistema proposto nesta tese pretende criar automaticamente animações a partir de um qualquer programa/algoritmo e de algumas indicações do utilizador sobre a forma de visualizar essa animação. Assim sendo, trata-se de um sistema do tipo IV.

2.5.1 Tipo 0

Relativamente ao tipo 0, serão apresentados alguns exemplos de sistemas que permitem a programação visual de animações, sem haver contudo qualquer geração automática de animações a partir de um texto fonte. É de notar que existem pelo menos duas espécies de sistemas pertencentes a este tipo: ambientes de programação visual que permitem observar o desenrolar da execução do programa (como o programa é composto por objectos gráficos, o que é observado é uma animação do programa); ou sistemas que

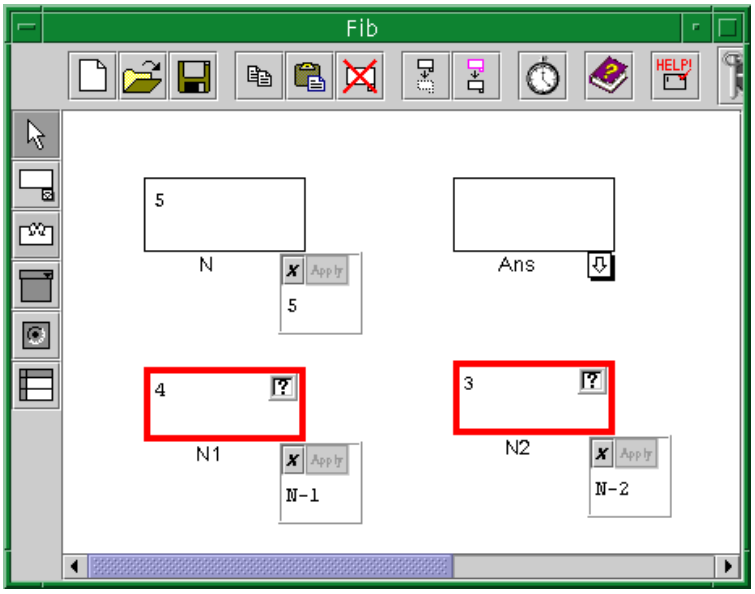

Figura 2.2: Animação do algoritmo Fibonacci

têm por objectivo a simulação de funcionamentos e que para tal usam também um conjunto de objectos gráficos que são animados consoante alguns parâmetros de entrada.

A linguagem visual FORMS3

O artigo [CBC96] descreve uma forma de integrar uma extensão visual e declarativa do paradigma *Path/Transition* numa linguagem visual de programação chamada Forms/3. Esta linguagem visual poderá ser usada para construir animações de algoritmos. Em Forms/3, o programador manipula directamente as células nos *forms* e define uma fórmula para cada célula. Cada fórmula pode incluir constantes, referências a outras células ou referência ao valor da própria célula no momento anterior. Os cálculos dos programas baseiam-se nestas fórmulas. A figura 2.2 representa um dos quadros de animação do algoritmo Fibonacci. Este quadro mostra a primeira iteração do algoritmo, onde Fibonacci de N é calculado com base em N-1 e N-2. O cálculo Fibonacci de N-1 e N-2 é feito em formulários semelhantes que representam as outras iterações deste algoritmo recursivo. O resultado final (`Ans`) será a soma das células `Ans` de todos os outros formulários.

A linguagem Forms/3 é declarativa e programar nesta linguagem é definir a relação entre entradas e saídas. Assim que a informação entra no sistema os efeitos reflectem-se imediatamente e o programador

apenas se preocupa com a especificação da animação.

Sob o ponto de vista do utilizador, a animação de algoritmos usa uma variedade de efeitos para comunicar a essência de um algoritmo: movimentos suaves, modificação gradual de cores e sequências de imagens. O utilizador intervém directamente na animação do algoritmo através de células com formulas e através de botões. Como já foi dito, esta abordagem segue o paradigma *Path Transition*. Neste modelo o caminho é definido por uma sequência finita de pares (x,y) de coordenadas. À travessia desse caminho chama-se transição.

Uma transição tem cinco parâmetros: objecto; caminho; tipo (movimento, intensidade, visibilidade e cor); evento de reset (restrições para o objecto reiniciar o seu caminho); evento de continuar (restrições para o objecto passar ao próximo passo do caminho). Embora um caminho de animação seja visto, muitas vezes, como uma sequência de pares (x,y) que o objecto percorre fisicamente, todos os tipos de transição são caminhos de animação de um objecto. Uma transição de intensidade anima o objecto ao longo das várias mudanças de intensidade.

Sob o ponto de vista do programador, a programação visual de animação consiste em especificar os parâmetros das transições dos objectos.

Um objecto poderá ser uma caixa, strings, objectos gráficos ou objectos complexos resultantes de outros cálculos. O caminho é definido pelo programador através do ponto inicial e o ponto final.

É possível combinar animações, diferentes tipos de animação podem operar num mesmo objecto ao mesmo tempo. Isto é possível fazendo com que a entrada de uma animação seja saída de uma outra animação.

Forms/3 é reactivo porque é dada ao programador um *feedback* visual sobre os efeitos da alteração do programa. O programador pode modificar as restrições que conduzem a animação do algoritmo ao longo da sua execução e recebe imediatamente *feedback* do efeito que essas alterações causaram.

A execução do programa pode ser vista passo a passo ou à velocidade normal.

A linguagem visual HANDMOVE

Vodislav em [Vod97] apresenta um modelo para animação de interfaces que permite programação visual — *Handmove*. Este trabalho tem como objectivo final a definição formal do *Handmove* como uma linguagem visual de programação.

O sistema apresentado usa representações visuais baseadas em caminhos (*Path-Transition Paradigm* tal

como o Tango). A animação é descrita visualmente desenhando os objectos gráficos e suas trajectórias (sequências de posições espaciais). A animação é obtida quando um objecto segue um determinado caminho, executando uma acção de transformação em cada posição. As trajectórias podem ser alteradas dinamicamente.

O sistema *Handmove* organiza a animação em cenas. Uma cena engloba um conjunto de objectos no mesmo espaço bidimensional. Uma cena pode receber estímulos de animação ou eventos de aplicações e do utilizador e pode despoletar eventos internos.

São utilizados objectos gráficos que são caracterizados por um nome, uma forma geométrica, um conjunto de atributos gráficos e um conjunto de atributos geométricos e formados por relações de composição. Composição define as restrições gráficas e espaciais entre o objecto e as suas componentes. Assim, poderá haver movimentos do objecto como um todo ou movimentos individuais de uma componente. Este último movimento será sempre relativo ao movimento do resto do objecto.

Neste sistema é também utilizado o conceito de *actor*. Um *actor* é um objecto gráfico dinâmico englobado numa cena. O seu comportamento é descrito pela sua trajectória abstracta (variação dos seus atributos gráficos e geométricos). Os *actores* reagem a estímulos de animação, sinais que produzem uma continua evolução. Assim, existem vários tipos de animação: espontâneas, interactivas e as controladas pela aplicação. As primeiras são controladas por estímulos de relógios internos e são usadas em apresentações animadas. As segundas usam os estímulos do rato ou outros eventos despoletados pelo utilizador, e as últimas usam animações pré-definidas despoletadas por sinais do algoritmo.

As trajectórias são compostas por *segmentos* e por *spots*. O link visual entre *actor* e a sua trajectória é feito associando a origem da trajectória com o nome do *actor*. Um segmento é uma parte homogénea da evolução e dois segmentos são sempre separados por um *spot*. Um *spot* representa uma mudança repentina de evolução, pertence sempre a um segmento (ao anterior), é definido por uma condição a verificar e por uma lista de acções a executar. O primeiro spot é a origem da trajectória e expressa a condição inicial e as acções iniciais para a animação.

As acções podem ser: mudanças de status, mudanças de evolução ou emissões de eventos. Um spot pode ser uma posição num segmento, um valor de um atributo, um momento no tempo, o valor de um estímulo de uma aplicação ou a recepção de um evento.

O modelo visual *Handmove* pode ser visto como a sintaxe de uma linguagem visual e o modelo de animação *Handmove* deverá ser visto como sendo a parte semântica dessa mesma linguagem. Para tal, serão usadas gramáticas relacionais (para a sintaxe visual) e gramáticas de atributos relacionais para

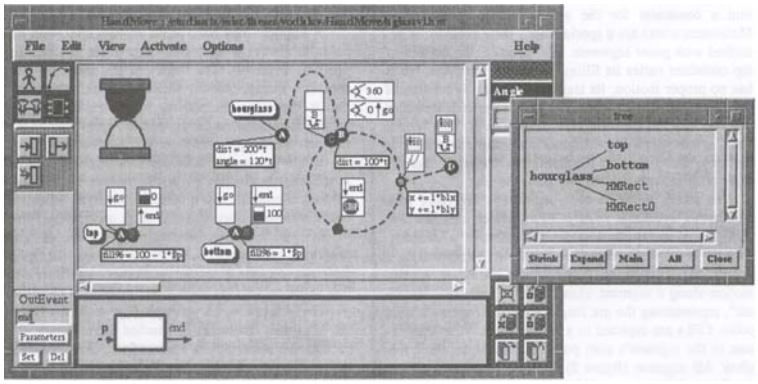

Figura 2.3: Programa visual da animação (Handmove)

especificação da semântica onde a cada produção será adicionado um conjunto de regras de cálculo de atributos.

A figura 2.3 apresenta um exemplo de utilização do sistema onde se pode ver o programa visual de uma cena da animação a produzir.

O Sistema AGG

O sistema AGG (Attributed Graph Grammar System) é um ambiente algébrico de programação baseado numa linguagem visual de *regras de produção* para transformação de grafos [MRRT99]. O comportamento do sistema é especificado por regras de reescrita de grafos usando uma descrição do tipo IF-THEN. Os grafos AGG podem ter atributos, declarados como tipos de objectos Java, associados aos nodos ou aos arcos.

A aplicação de uma regra de reescrita transforma a estrutura do grafo. A aplicação sequencial de várias regras mostra o cenário da aplicação.

O sistema AGG pode ser usado para especificar e prototipar a implementação de aplicações cujas estruturas de informação são grafos.

O ambiente de desenvolvimento contém editores gráficos (para grafos e regras) e um editor textual (para expressões Java a incluir nas regras desenhadas no editor visual). Este sistema suporta interpretação visual.

Uma gramática de grafos contém um grafo inicial e um conjunto de regras de produção que podem ter condições para que não seja possível a sua aplicação (Negative application conditions).

Um grafo consiste num conjunto de nodos e arcos. Os nodos e os arcos são os objectos do grafo. Cada

arco representa uma ligação directa, orientada, de dois nodos (o nodo inicial e final do arco). Para uma melhor descrição de cada objecto do grafo podem ser incluídas etiquetas (labels). Este sistema permite múltiplos arcos do mesmo tipo entre dois nodos porque cada arco tem uma identidade própria. A posição de um nodo ou de um arco no plano não acarreta informação sintáctica, nem semântica. Ou seja, a disposição dos objectos do grafo é apenas uma questão de apresentação.

Um atributo é declarado como uma variável convencional e a sua declaração faz parte da definição de um tipo de nodo, ou de arco..

Uma acção pode ser vista como uma transição de estados e esta deve ser especificada através de descrições dos estados inicial e final. Assim, uma transição é descrita por dois grafos, um representa o estado antes da acção e o segundo o estado depois da acção.

O lado esquerdo das regras corresponde a todas as condições que devem ser satisfeitas para que a acção / transição possa ocorrer. Para indicar que um objecto do lado esquerdo corresponde a um lado direito são usadas etiquetas numéricas.

As regras são aplicadas pela ordem pela qual surgem na árvore apresentada no lado esquerdo do editor. Cada regra é aplicada tantas vezes quantas for necessário antes de se verificar se se pode aplicar a regra seguinte. Sendo aplicada a última volta-se à primeira até que nenhuma possa ser aplicada.

O lado esquerdo da regra pode apenas conter constantes ou variáveis e nunca expressões. O lado direito pode conter expressões que usam variáveis do lado esquerdo e nada mais: assim, pode-se especificar a transformação semântica associada a cada transição.

As figuras 2.4 e 2.5 mostram dois exemplos de programas. O primeiro mostra a especificação de três regras onde se define o conceito de *homem*, de *mulher* e de *casamento*. O *casamento* é efectuado apenas entre um *homem* e uma *mulher* e ambos têm que ser solteiros. Na última parte da figura surge o programa de entrada ao qual vão ser aplicadas as regras.

O segundo exemplo pretende simular o funcionamento de uma máquina que vende cigarros. A máquina tem a seguinte informação associada: preço por maço, dinheiro em caixa e stock de maços. Existem ainda outras duas entidades que se relacionam com a máquina, o cliente e o dono da máquina. A primeira regra representa a alteração de preço por maço; a segunda especifica a acção de retirar o dinheiro em caixa; a terceira simula uma compra tendo em conta o dinheiro do cliente e o número de maços em stock; e, a última regra representa o abastecimento da máquina. Na última parte da figura pode visualizar-se um exemplo de um caso concreto onde serão aplicadas estas regras de modo a calcular-se a situação final.

Numa nova versão deste sistema é possível definir a relação entre as várias variáveis usadas para calcular

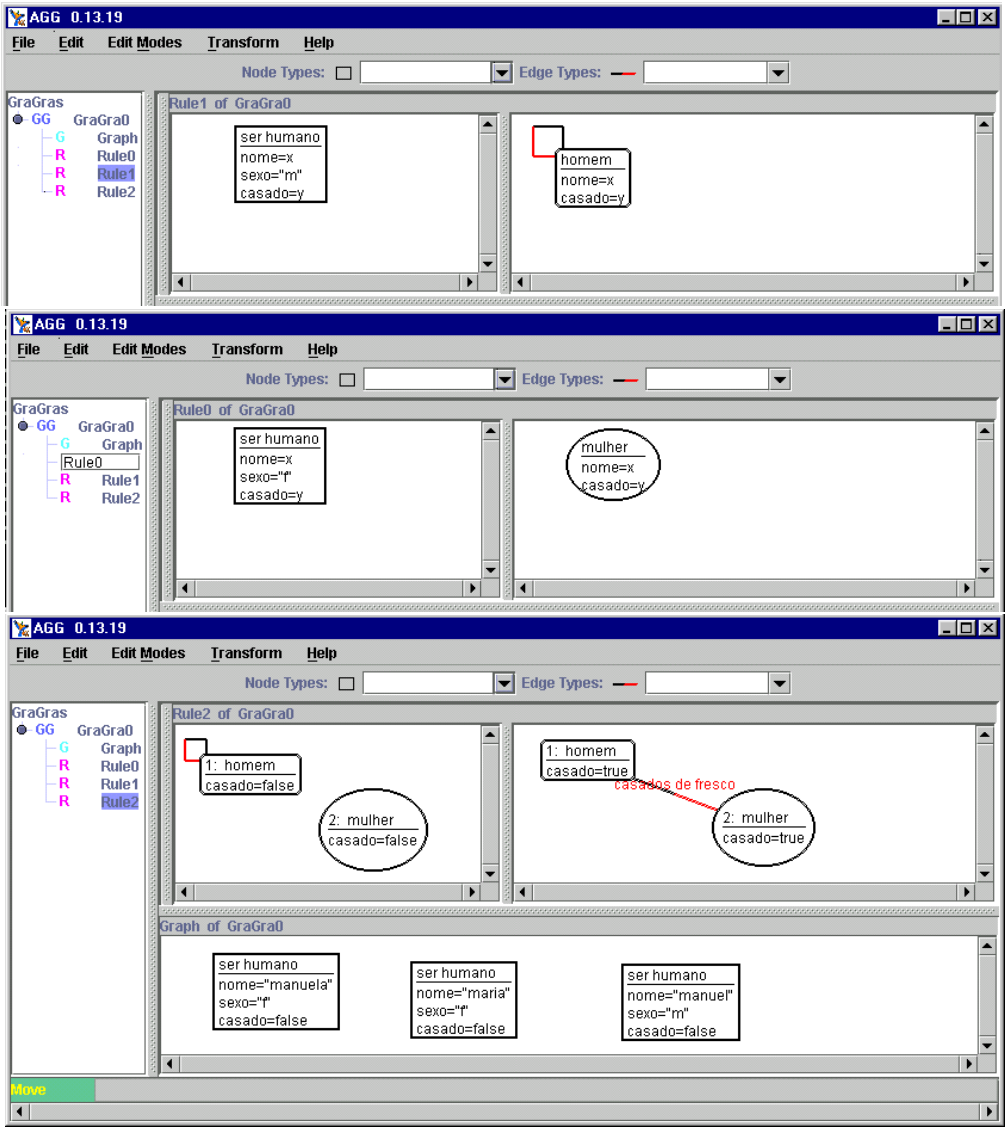

Figura 2.4: Primeiro exemplo de um programa AGG

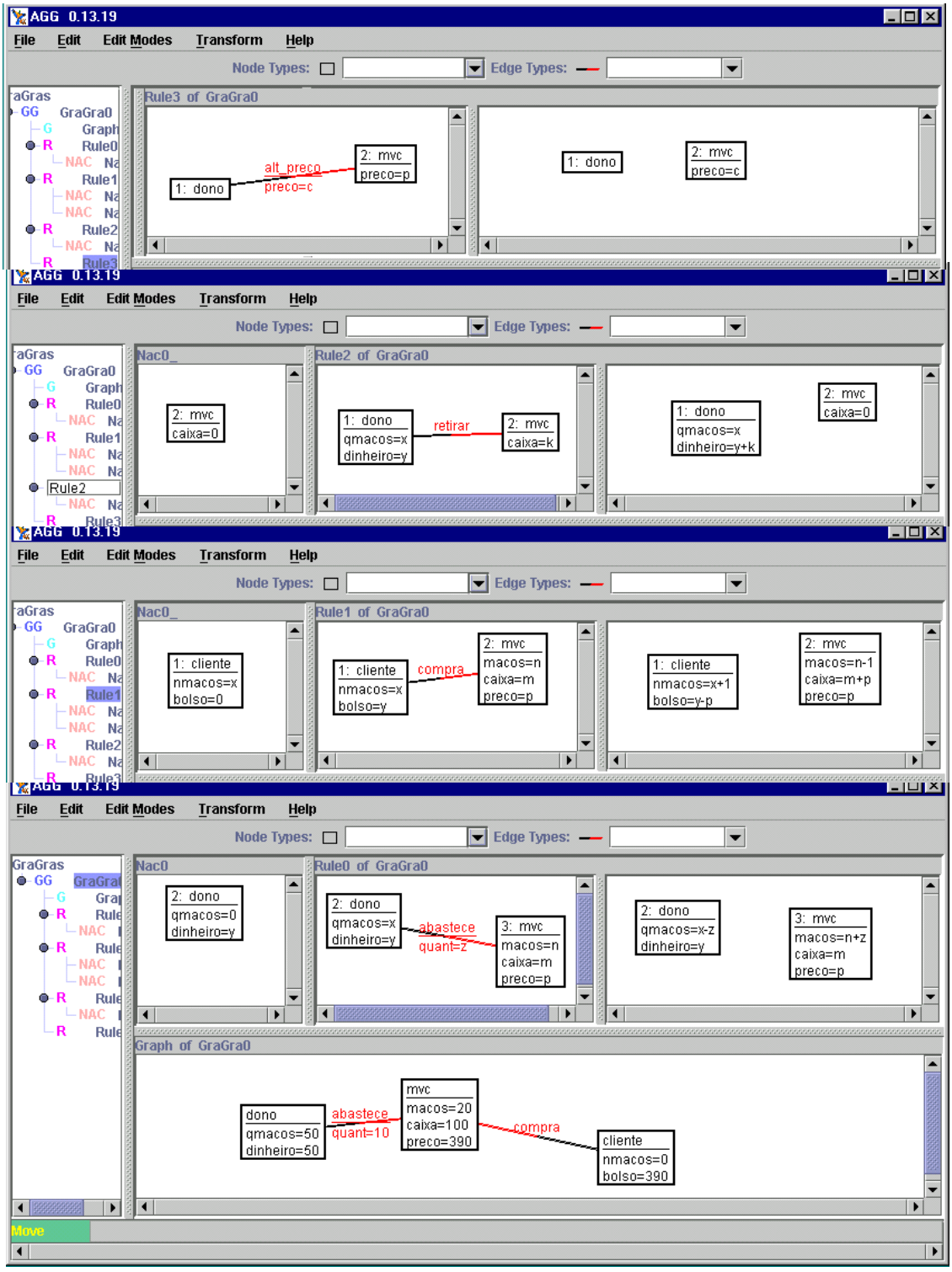

Figura 2.5: Segundo exemplo de um programa AGG

o valor dos atributos. Ou seja, é possível obrigar a que dois objectos do mesmo tipo ou de diferentes tipos tenham o mesmo valor para o mesmo atributo. As figuras G.1, G.2, G.3 e G.4, inseridas no apêndice G, mostram dois programas com essas características.

Da experiência adquirida através da construção destes exemplos, surgiram algumas notas importantes. Quando uma regra obriga a que um nodo desapareça os arcos que têm o nodo eliminado como nodo fonte ou nodo destino deixam de fazer sentido e também desaparecem. No desenho do grafo inicial não é possível usar variáveis nem expressões associadas aos atributos. Por último, é de salientar que o facto de definir atrib=x em dois nodos diferentes do mesmo tipo, não obriga a que tenham o mesmo valor (x) para o mesmo atributo (atrib).

Para concluir, o sistema AGG permite programar alterações a que um determinado grafo (constituído por um conjunto de objectos, possivelmente de tipos diferentes, relacionados ou não entre si) irá ficar sujeito durante uma simulação do funcionamento de algo que representa. Por outro lado, se esse algo for um programa visual, a simulação do seu funcionamento será uma animação de programa. Uma animação não será mais do que a visualização da reescrita do cenário inicial até ao cenário final. Dependendo do tipo de linguagem usada no programa que se pretende animar, pode ser ou não mais fácil fazer o mapeamento entre o funcionamento que se pretende simular e as regras de reescrita de grafos usadas no sistema AGG. Estas considerações permitiram o nascimento de algumas ideias que serviram de base à concepção da arquitectura do sistema proposto nesta tese.

O sistema FRAN (breve referência)

Em [Ell98], Conal Elliot apresenta a abordagem subjacente ao sistema FRAN, que produz animações funcionais reactivas. FRAN é um vocabulário de alto nível que permite descrever as características principais de um modelo animado, omitindo detalhes de apresentação.

Este vocabulário está integrado numa linguagem de programação funcional moderna: Haskell. Os modelos de animação assim descritos serão reutilizáveis e compostos das mais diversas formas.

O modelo CPN (breve referência)

Kurt Jensen em [Jen96b] descreve um sistema de modelação e simulação do funcionamento de sistemas baseado em *Redes de Petri*. Uma *Rede de Petri* é uma rede de nodos e actividades inter-conectadas,

com regras que determinam o momento em que uma actividade pode ocorrer e especificam como a sua ocorrência modifica o estado do nodo associado a essa actividade.

As *Redes de Petri* podem ser usadas para modelar e simular sistemas de qualquer tipo. Existe um formalismo matemático associado às *Redes de Petri* que define o que são, e como se comportam.

Normalmente, uma *Rede de Petri* é representada por um grafo, no entanto, actualmente, é apenas um objecto matemático que existe independentemente da sua representação física. A *Rede de Petri* foi evoluindo, ao longo do tempo, para um formalismo mais complexo: *Rede de Petri Colorida Hierárquica* (RPC) para representar modelos com uma estrutura hierárquica, suportando informação de diferentes tipos e valores.

O modelo RPC [Jen96a] serviu de base para a criação de uma ferramenta interactiva (Design/CPN) para modelação e simulação com *Redes de Petri*. A ferramenta inclui um editor para criar e manipular as RPCs; um verificador de sintaxe para as validar; um simulador para as executar; capacidades de *debugging* e monitorização interactiva; facilidades de organizar a rede em módulos hierárquicos; facilidades de animação para apresentação dos resultados da simulação. Estas capacidades permitem criar, modificar, organizar, executar, deporar (debugger), examinar e validar os modelos de sistemas baseados em RPC.

Tal como o Design/CPN, muitos outros ambientes de especificação têm como objectivo a modelação e simulação do funcionamento de sistemas. Considera-se que o sistema permite a programação visual de objectos gráficos de modo a que a sua execução gere uma animação capaz de reproduzir comportamentos.

O projecto VCG (breve referência)

Existem outros sistemas que, tendo objectivos semelhantes aos anteriores, requerem apenas especificações textuais para gerar as visualizações (animações estáticas) de estruturas. É o caso do projecto VCG (*Visualization of Compiler Graphs*) que pode ser consultado em [San95].

A ferramenta VCG recebe especificações textuais de um grafo e produz uma visualização desse grafo. É usada, então, uma linguagem que permite descrever textualmente o grafo e o aspecto dos nodos e dos arcos — GDL (Graph Description Language). O principal objectivo deste sistema é produzir visualizações das representações intermédias usadas nos compiladores de forma rápida e eficiente. Se as posições dos nodos não forem fixas, o sistema produz o desenho do grafo, usando várias heurísticas, para reduzir o número de cruzamentos de arcos, minimizar o tamanho dos arcos e centrar as posições dos nodos.

O sistema *Magic Animator*

Magic Animator [Nor94] é uma ferramenta para Windows que usufrui dos menus habituais deste sistema e cujo objectivo é *mostrar filmes* (sequência contínua de imagens). É um sistema que permite criar imagens e mostrá-las numa determinada sequência de modo a ser produzida uma animação. Cada imagem é constituída por desenhos a duas dimensões, efectuados pelo utilizador com a ajuda de ferramentas de desenho.

2.5.2 Tipo I

Nesta secção irão ser apresentadas animações que ilustram o Tipo I. Muitas delas foram encontradas no *site* [BV99] que contem uma colecção bastante completa de animações criadas para algoritmos específicos; na maior parte das vezes, não é mencionada a linguagem usada para implementar esses algoritmos.

Animação do algoritmo Bubblesort

A animação do algoritmo apresentado em [Hau99a] dá a conhecer um tipo de visualização de um algoritmo de ordenação. Neste exemplo, o utilizador pode controlar a velocidade de execução da animação, é usado um gráfico para visualizar os valores a serem ordenados e são usadas cores para distinguir os elementos envolvidos em cada fase do algoritmo. No entanto, este tipo de animação serve apenas para este algoritmo e não permite que seja o utilizador a indicar os valores a serem ordenados.
A figura 2.6 representa a animação do seguinte algoritmo (numa variante do *bubblesort* que não pára quando já não há trocas):

```
for (i=0; i<n-1; i++) {
  for (j=0; j<n-1-i; j++)
    if (a[j+1] < a[j]) {  /* compara os dois vizinhos */
      tmp = a[j];         /* troca a[j] com a[j+1]      */
      a[j] = a[j+1];
      a[j+1] = tmp;
  }
}
```

O algoritmo começa com um array de números aleatórios. Cada número é representado por um traço vertical. A altura do traço representa o seu valor e a sua posição no gráfico indica a posição no array.

Animação do algoritmo de Graham's Scan

Da mesma família da animação apresentada no exemplo anterior existem outras animações em [Hau99b] que não estão relacionadas com algoritmos de ordenação. O algoritmo de *Graham's Scan*, dado um conjunto de pontos num plano, calcula o seu fecho convexo, ou seja, o polígono com o menor número de lados que contém um determinado conjunto de pontos. Em primeiro lugar procura um ponto extremo (ponto com maior coordenada y - *pivot*), de seguida procura um outro ponto que forme um maior ângulo em relação ao *pivot* e assim sucessivamente até que esteja construído um polígono que englobe todos os pontos do plano.
Para este algoritmo, apresenta-se algumas imagens da respectiva animação na figura 2.7.

Animação de algoritmos matemáticos

Existem também animações em [CSC94] relacionadas com o ensino da matemática. A título de exemplo são mostradas na figura 2.8 algumas imagens pertencentes a um vídeo que ilustra o fenómeno Gibbs numa série de Fourier. Este tipo de animação, por ser um video, não permite qualquer intervenção do utilizador.

Animações criadas no Ambiente Leonardo

O sistema Leonardo, que irá ser apresentado na secção 2.5.4, permite construir animações como as que são mostradas em [DF01a]: são animações de algoritmos conhecidos que não permitem a sua parametrização (usar o mesmo algoritmo com valores diferentes) nem qualquer outro tipo de intervenção por parte do utilizador final. Apresenta-se como exemplo as imagens da figura 2.9.

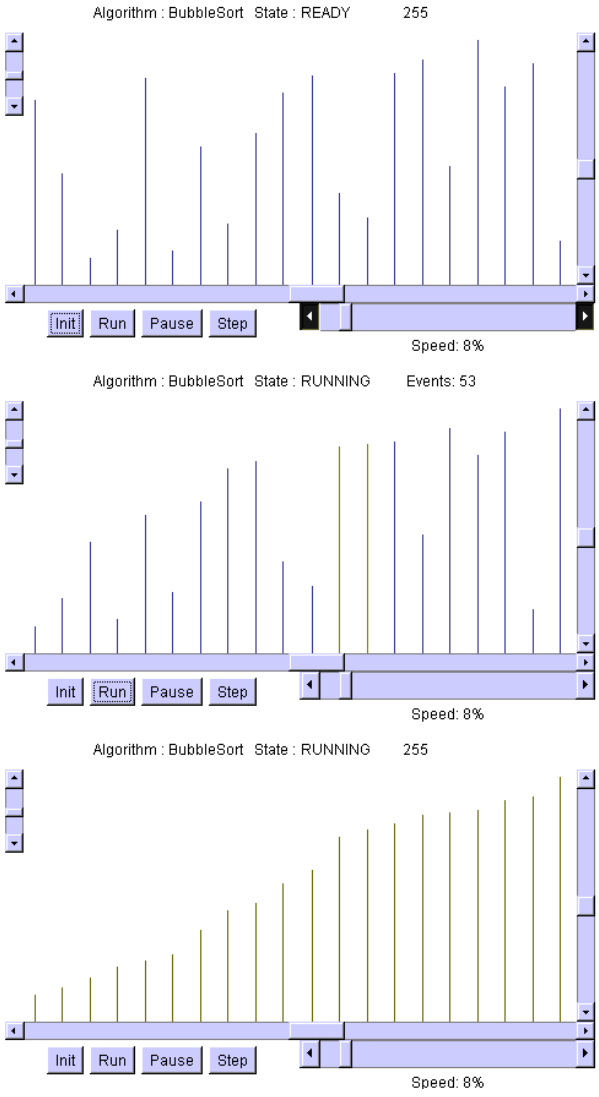

Figura 2.6: Algoritmo Bubblesort

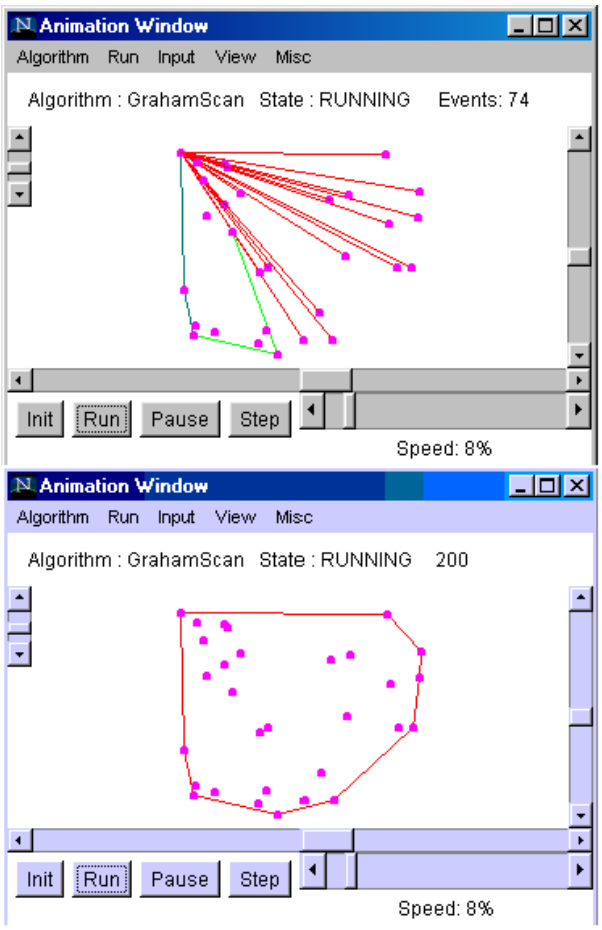

Figura 2.7: Algoritmo Graham's Scan

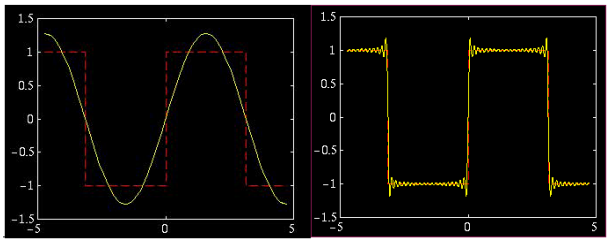

Figura 2.8: Fenómeno de Gibbs numa série de Fourier

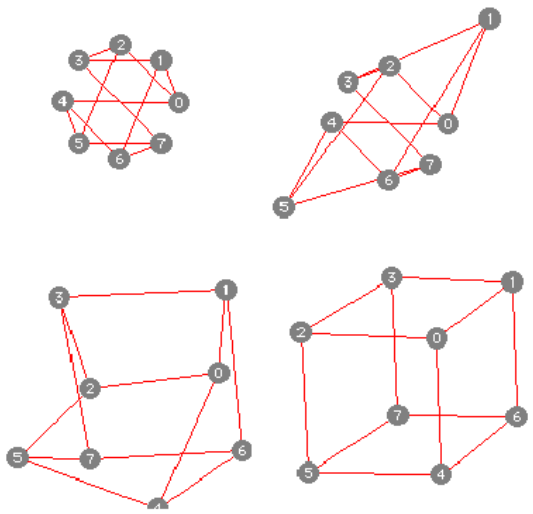

Figura 2.9: Animação de grafos

2.5.3 Tipo II

As animações apresentadas nesta secção correspondem ao Tipo II e, por isso mesmo, tornam-se mais interessantes e proveitosas para os seus utilizadores. O facto de poder intervir no funcionamento do algoritmos através da inserção de valores e até, poder escolher outros algoritmos dentro do mesmo estilo de animação permite perceber com mais facilidade como estruturar e esquematizar a solução de problemas.

Animação de programas C

Relativamente ao uso de animação de algoritmos no ensino da programação em C existem também alguns exemplos interessantes em [Dav94] como o representado na figura 2.10. Esta animação, permite que o utilizador crie uma lista e indique os elementos que pretende inserir ou remover da lista criada. Esta animação permite uma maior interacção com o utilizador do que as animações apresentadas anteriormente. Ou seja, embora esta animação não permita alterações do algoritmo nem da forma de visualizar a sua execução, o utilizador pode repetir a simulação quantas vezes pretender aplicada a valores de teste diversos.

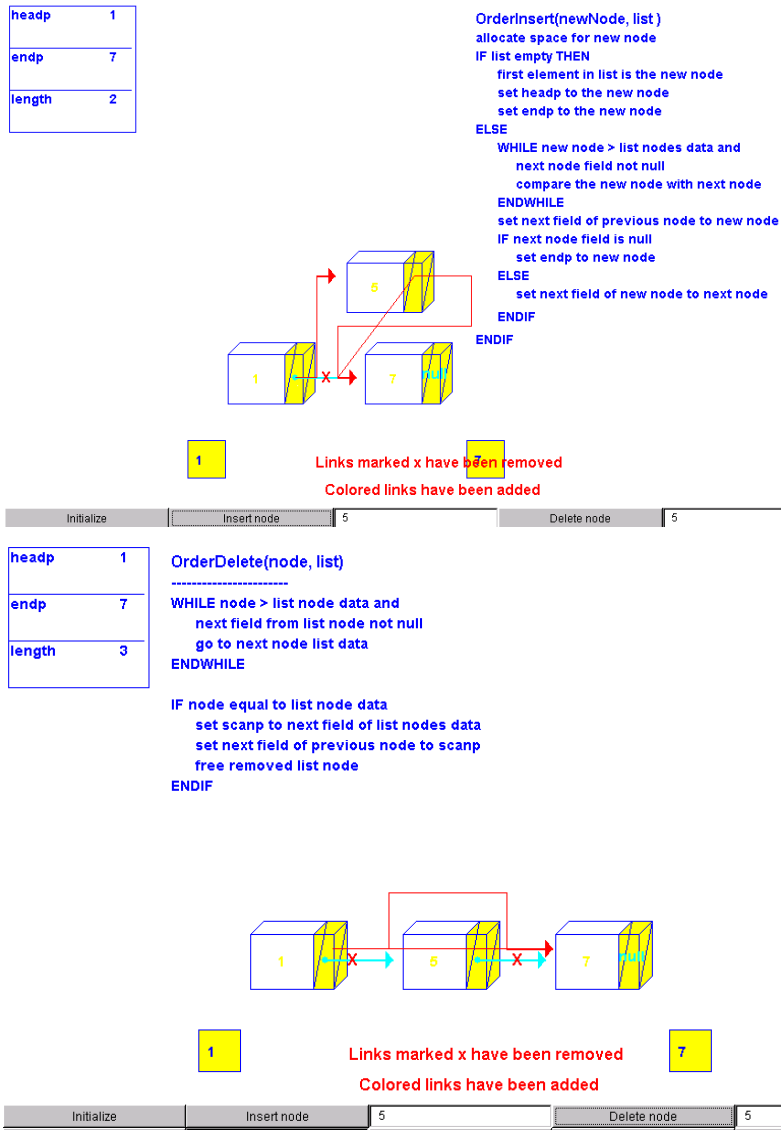

Figura 2.10: Inserção e remoção de elementos de listas em C

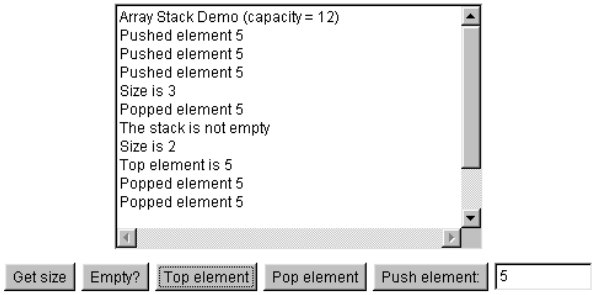

Figura 2.11: Animação de pilhas

Animação de Pilhas

A animação que vai ser apresentada é um pouco mais interactiva do que as anteriores e foi recolhida em [Goo99]. Curiosamente, a interface é textual e não gráfica como as demais. O *applet* Java apresentado permite não só inserir e remover elementos da pilha como também obter informações sobre o estado da pilha.

Embora esta animação seja mais manuseável é especifica para a manipulação de pilhas. Uma imagem desta animação pode ser vista na figura 2.11.

Visualizações interactivas de estruturas de dados

No âmbito do projecto JCAT, Duane J. Jarc apresenta em [Jar97] várias visualizações de estruturas de dados que permitem um certo grau de interactividade. Estas visualizações baseiam-se em exemplos já preparados e mostram o funcionamento de uma operação (escolhida pelo utilizador de entre várias) aplicada a esses valores de forma animada.

Na figura 2.12 é apresentado um exemplo de procura de um elemento num grafo.

2.5.4 Tipo III

Relativamente a sistemas de animação do Tipo III, multi-algoritmo (mas mono-linguagem), existem alguns exemplos.

Os primeiros sistemas de animação recorriam a anotações de animação acrescentadas ao algoritmo de entrada sendo este algoritmo descrito numa determinada linguagem. Nesta secção vão ser apresentados alguns sistemas que recorrem a outras abordagens como é o caso do sistema VIP, LENS e ZSTEP. Mais

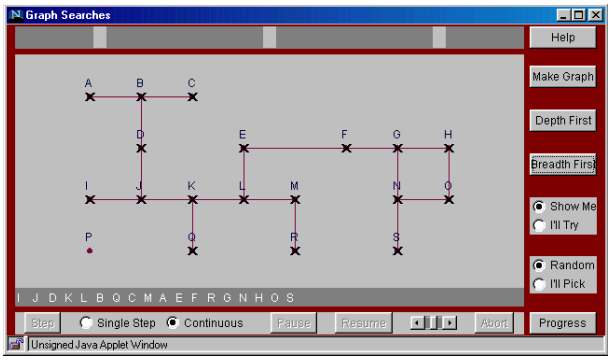

Figura 2.12: Procura de elementos em grafos

recentemente surgiram, deste mesmo tipo, os sistemas JELIOT, JCAT e JAWAA, que também irão ser apresentados nesta secção.

Os primeiros sistemas de animação

BALSA (Brown ALgorithm Simulator and Animator) ([BS84] citado em [SDBP97]) foi o primeiro sistema que permitiu a programação textual de animações de algoritmos; surgiu em 1981, e tornou-se num modelo seguido nos trabalhos posteriores. Neste sistema alguns pontos estratégicos do algoritmo a animar eram anotados com chamadas a procedimentos de animação.

Em 1987, surgiu o sistema ANIMUS ([Dui98] citado em [SDBP97]) que, sendo também um sistema de programação textual de animações, inclui restrições temporais na construção de animações de algoritmos. Cada objecto pode ter uma representação gráfica que é automaticamente actualizada em resposta a alterações do objecto e animador pode especificar restrições nas relações entre esses objectos.

Mais tarde, em 1989, apareceu o sistema ALADDIN ([HHR89] citado em [SDBP97]) que permite especificar visualmente a animação de programas textuais.

TANGO [Sta90] surge em 1990 com um modelo de animação com uma semântica precisa suportado pelo paradigma Path Transition. Este paradigma baseia-se no movimento contínuo e suave de uma imagem, considerando conceptualmente todos os tipos de animação como uma imagem ao longo de sucessivas alterações. O sistema TANGO cria a animação indicando passo a passo as modificações a efectuar. Existe também o sistema XTANGO [Sta01] que permite construir animações coloridas em tempo real. Neste sistema, a construção da animação consiste em: implementar o algoritmo a animar em C; decidir quais os eventos importantes (instruções significativas), ou seja, aqueles a serem retratados durante a execução

do algoritmo e marcar esses eventos. Estes eventos activam rotinas de animação que são implementadas num ficheiro à parte; as transições dos objectos incluem movimento, mudança de cor, de tamanho ou de conteúdo.

A seguir ao sistema XTANGO surgiu o sistema POLKA [Sta99a] (sistema para construir animações) que, sendo mais poderoso e flexível, tem vindo a ser actualizado. Os autores deste sistema (tal como a maior parte das pessoas que trabalham nesta área) baseiam-se na ideia que a animação de um algoritmo é uma representação gráfica e dinâmica de dados e operações que tornam o algoritmo mais concreto e mais fácil de entender. Para facilitar o uso deste sistema foi incluído um *front-end* chamado SAMBA. Enquanto os sistemas apresentados atrás são utilizados por alguém que quer explicar algoritmos, o SAMBA tem por objectivo permitir que sejam os próprios alunos a criar animações gerando funções de visualização. SAMBA [Sta99b] é um interpretador interactivo de animações que lê comandos ASCII (para definição de formas geométricas, coordenadas, movimentos do cursor e cores), e produz acções de animação. Assim, será fácil anotar um programa escrito em qualquer linguagem para gerar esses comandos e a partir destes obter funções de animação.

O sistema POLKA, tal como o anterior (XTANGO), obriga a que o código fonte seja alterado para que o programa possa ser animado. A animação é programada textualmente (através de anotações), associando a cada anotação desenhos e imagens que se julguem necessárias.

Em 1993, surge um sistema chamado LENS ([MS93] citado em [SDBP97]) que permite ao programador desenvolver animações de programas textuais sendo toda a especificação e desenho feitos num ambiente visual.

Outros dois trabalhos usam técnicas de visualização para fazer o debug de linguagens textuais: PROVIDE ([Moh88] citado em [SDBP97]) e ZSTEP [LF95].

Alguns sistemas de visualização baseiam-se numa abordagem declarativa. Esta abordagem tornou-se muito atractiva porque as visualizações podem ser especificadas e modificadas facilmente e porque a visualização não está associada directamente ao código fonte. O sistema Pavane [RCWP92] emprega noções de visualização declarativa, estabelecendo esta abordagem como sendo um novo paradigma. Mas não foi o primeiro sistema a basear-se nestas noções: o sistema PROVIDE, já referido anteriormente, especifica a visualização criando *mappings* entre as variáveis do programa e icons que representam os seus atributos. O sistema ALLADIN, também ele já referido, emprega uma abordagem declarativa para definir o relacionamento entre as variáveis e as suas imagens. No entanto, este sistema combina esta especificação com uma outra abordagem por anotações, que indica quais os pontos do programa onde

cada imagem será alterada.

Todas as referências apresentadas nesta secção dizem respeito a sistemas de criação de animações de programas. Existem, no entanto, outros trabalhos relacionados com a aplicação deste tipo de sistemas ao ensino da programação. Alguns exemplos podem ser encontrados em: [McW96], [Mic96], [Sta96] e [SBC96].

Ainda nesta secção irão ser apresentados com mais detalhe alguns dos sistemas aqui referidos e outros sistemas mais recentes que implementam algumas das técnicas faladas na secção anterior.

O Sistema VIP

VIP [MM88] é uma ferramenta para visualizar conteúdos de variáveis e informações sobre o controlo de fluxo do programa. Trata-se de um software educativo na área do desenvolvimento de algoritmos e tem como objectivo ajudar os alunos a compreender os mecanismos básicos da programação estruturada. Os alunos escrevem algoritmos numa linguagem pré-determinada, mas bastante simples, e usam o sistema VIP para seguir, de forma dinâmica, a sua execução.

Durante a simulação os alunos podem ver a cada momento qual a instrução que está a ser executada, de que forma esta afecta o valor das variáveis, o *output*, e como cada estrutura de controlo afecta a sequência do programa. Este sistema apresenta alguns pontos positivos na medida em que permite repetir a simulação da execução do algoritmo quantas vezes for necessário e permite também motivar os alunos para a programação.

No desenvolvimento do sistema VIP foi necessário adoptar uma linguagem a ser usada na especificação de algoritmos. A especificação de um algoritmo é feita de forma muito interactiva, usando comandos como: LE, ESCREVE, MUDALN, FAZ, SE, REPETE e FIM.

Durante a escrita de um algoritmo, quando um comando é indicado, o programa pede ao utilizador toda a informação necessária para a sua execução. Logo que o algoritmo é especificado, o programa verifica erros e informa o utilizador caso os tenha detectado. Durante a execução do algoritmo o ecrã divide-se em várias áreas: área do algoritmo, área das variáveis, área de input/output e a área de comentários. A execução pode ser feita passo a passo ou de forma contínua.

O sistema VIP foi desenvolvido em Pascal e tem a interface mostrada na figura 2.13. O sistema verifica se todos os comandos usados são válidos e se todas as variáveis usadas foram previamente declaradas. Dependendo do comando que está a ser executado várias operações devem ser efectuadas. Nas instruções

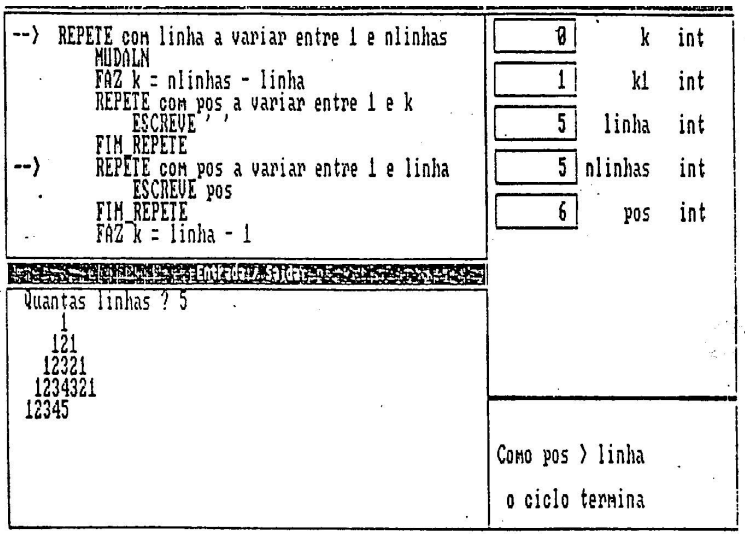

Figura 2.13: Sistema VIP

de selecção e de repetição é necessário calcular o resultado de uma expressão e depois executar um conjunto de outros comandos dependendo do valor calculado.

O sistema LENS como exemplo de manipulação directa de visualizações

A parte gráfica de uma visualização pode ser mais ou menos sofisticada, assim como, poderão ser usadas bibliotecas de primitivas de desenho para esse efeito. No entanto, essas bibliotecas são úteis principalmente para domínios específicos e implicam uma certa demora no desenvolvimento das visualizações. Por isso, surgiram umas ferramentas que permitem criar de forma fácil e visual as animações: sistemas baseados em manipulação directa das visualizações. Um exemplo dessas ferramentas é o sistema LENS. LENS ([MS93] citado em [SDBP97]) é um debugger visual para programas C e funciona em plataformas Unix e XWindows. Este sistema usa janelas para introduzir o código fonte; para desenhar a aparência dos objectos, e para apresentar os comandos de debug.

O programador pode recorrer aos comandos normais de debug ou a comandos especiais de animação, associados a certos pontos do programa. O programador escolhe as variáveis e as posições no texto do código fonte e, não se preocupando com o uso de primitivas de gráficas, desenha no editor visual a aparência dessas variáveis nas respectivas posições no programa. As figuras ficam então associadas a posições do código fonte e sistematizando a construção das figuras para todas as instruções, obteremos a

visualização de todo o programa. O utilizador, a qualquer momento pode alterar a animação, alterando as figuras associadas a cada objecto do programa.

O sistema ZSTEP

O sistema ZSTEP [LF95] é um *debugger* reversível que cria uma visão animada do programa. É um ambiente de deporação de programas que permite ao utilizador entender a correspondência entre o código estático de um programa e a sua execução dinâmica. Este sistema cria uma visão animada do programa fonte; cria uma janela com os valores das variáveis e o historial do programa (gerado incrementalmente); contém controlos de execução do programa (sentido da sequência de animação e nível de detalhe); permite acesso rápido a partir de uma expressão aos seus valores e à sua visualização.
O sistema ZSTEP é visto como sendo do tipo III, na medida em que obriga o utilizador a usar um editor e uma linguagem próprios do sistema.

O sistema JELIOT

No âmbito do trabalho descrito em [HPS+97] foi apresentado um ambiente chamado JELIOT, que permite ao utilizador animar os seus próprios algoritmos bastando, para isso, seleccionar os objectos que pretende animar. A animação é, então, gerada automaticamente.
Tradicionalmente, a animação era construída inserindo chamadas a primitivas de animação ao longo do código. A animação no ambiente JELIOT é controlada através de operações de tipos de dados, não havendo código adicional.
O código Java fica sujeito a uma pré-compilação que consiste em extrair variáveis do algoritmo que poderão ser animadas e formar uma lista estruturada. O gerador de código substitui as operações por chamadas a métodos que, por sua vez, são vistos pelo animador como chamadas de animação. Depois da pré-compilação o applet Java é compilado normalmente.
A interface usada neste ambiente permite que o utilizador defina a aparência visual da animação. Cada animação é composta por vários *actores*. Cada *actor* é uma entidade gráfica, com atributos visuais (tamanho, cor, localização), que representa um objecto de informação e, como tal, tem um *papel* a desempenhar na animação. Assim, cabe ao utilizador definir a aparência de cada *actor*. O *director* faz a

gestão dos vários estados de animação. Um estado pode ter vários *actores* (entidades) e um *actor* pode estar em vários estados nos quais se pode apresentar de forma diferente.
A representação visual de uma variável é um objecto que contem o valor da variável e será etiquetado com o nome dessa variável. A aparência das variáveis pode ser seleccionada a partir de algumas visualizações pré-definidas. As operações fazem mover esses objectos e mostram os respectivos operadores. Um objecto torna-se auto-animado quando é substituído por uma instância de uma classe de animação. Existem dois tipos de classes de animação: umas que se relacionam com os *actores* e outras que se relacionam com os seus *papeis*. Por exemplo, uma variável poderá ser uma instância da classe *papel* que, por sua vez, inclui métodos visuais que criam instâncias de classes *actor*.
O *director* manda pedidos de animação aos seus *actores*. O animador inclui os estados de animação do *director*, a comunicação entre estes e a interface para o utilizador. O animador anima uma determinada operação enquanto o applet de animação fica bloqueado e depois devolve-lhe o controlo.
Este sistema de animação permite programar animações de programas textuais sem alterar o programa, no entanto, obriga a usar tipos de dados especiais (o que implica uma pré-compilação). O sistema proposto nesta tese não implica qualquer tipo de pré-compilação; também não requer a alteração do código fonte.

O sistema JCAT

O artigo [BNR97] fala-nos de um sistema interactivo de animação de algoritmos e foca a comunicação entre professores e alunos explorando o modelo cliente/servidor. O sistema apresentado baseia-se no modelo BALSA sendo alguns pontos estratégicos do algoritmo anotados com chamadas a procedimentos criando os chamados *eventos interessantes*.
Cada vista responde a cada evento desenhando as imagens apropriadas. Assim, preparar uma página de *textbook JCAT* consiste:

- definir os eventos interessantes (usando uma interface em Java)
- implementar o algoritmo e anotá-lo (applet com métodos de implementação)
- implementar uma ou mais vistas (applet com métodos de desenho)
- criar páginas web que usem o algoritmo e as vistas

Este sistema obriga a que o código fonte seja alterado para que o programa possa ser animado. Neste caso, a animação é programada textualmente (através de anotações), associando a cada anotação desenhos e imagens que se julguem necessárias.
A figura 2.12, já apresentada, é um exemplo de uma animação criada por este sistema.

O ambiente LEONARDO

O sistema LEONARDO[DF01b] é um ambiente integrado para desenvolvimento e animação de programas em C. Foi criado especificamente para: suportar a aprendizagem da linguagem C; efectuar *debug* visual de programas C; criar animações de algoritmos num estilo declarativo.
O sistema permite editar, compilar e executar programas em C. É também possível inverter, a qualquer momento, a execução do programa. Os programas podem ser animados anotando o código C com declarações especiais, escritas numa linguagem simples, chamada ALPHA, pertencente ao paradigma lógico.
As animações após terem sido geradas e quando estão a ser apresentadas não permitem qualquer tipo de intervenção do utilizador. Foi nesta perspectiva que foi incluído um exemplo destas animações na secção 2.5.2 deste mesmo capítulo.

O sistema ZAL

Em [OPMS98] é descrito um trabalho cujo objectivo é facilitar o processo de validação do programador, animando especificações Z. Esse objectivo é conseguido usando um editor *TranZit* que também verifica os tipos e a sintaxe da linguagem na construção de especificações Z; um motor de transformação que gera a representação executável da especificação Z para animação no formato Lisp extendido; o sistema ZAL, que é um ambiente de animação baseado em Lisp, produz o mecanismo de execução da animação; e, por último, um sistema de visualização chamado *ViZ* que permite a compreensão, clarificação e validação da especificação formal executável, permite ao utilizador escolher a representação apropriada para cada objecto e criar animações estáticas e dinâmicas desses objectos de uma forma interactiva e iterativa e produz um modelo de visualização genérico que captura o comportamento dinâmico e estático da especificação ZAL.
O sistema ZAL executa animações com base na especificação executável que lhe é passada e permite aos programadores interactuar com essas animações com o objectivo de demonstrar as várias propriedades

da especificação original.

O Sistema SICAS

O sistema SICAS [GM00] é um ambiente que possibilita, essencialmente, dois tipos de cenários: edição/resolução do problema e execução/simulação de problemas previamente construídos. No primeiro, o utilizador pode construir algoritmos através de representações visuais — fluxograma — recorrendo a simbologia gráfica que representa as principais estruturas necessárias à construção de um algoritmo. No segundo modo, o utilizador pode simular/animar a execução das resoluções construídas, analisando com o detalhe e ritmo desejado as várias fases e entidades no problema em causa.
Este sistema pertence ao tipo III porque usa uma linguagem específica, própria do sistema, através de um editor gráfico e usa essa representação para criar simulações de comportamentos.

Sistema de criação de animações em Java

Existem outras animações interessantes em [Rod96]. Estas animações pretendem explicar o funcionamento de algoritmos conhecidos e foram desenvolvidas no projecto JAWAA. Um exemplo pode ser visto nas figuras 2.14 e 2.15, que mostram de forma elucidativa o funcionamento de um parser. Para uma determinada gramática fixa é mostrada a tabela de parsing e a string (também fixa) que está a ser reconhecida.
Uma vez feita a animação utiliza-se sempre os mesmos exemplos e o utilizador apenas controla a velocidade de execução.

Estes exemplos são fáceis de usar e úteis para quem pretende explicar os algoritmos subjacentes. Exemplos como este (do *parser*) podem ser programados através de uma interface, na qual os utilizadores descrevem as animações numa linguagem muito simples. O sistema JAWAA produz, a partir desse código, um applet Java que implementa os efeitos visuais pretendidos. A título de exemplo, apresenta-se no apêndice H, o código relativo a um algoritmo de travessia Depth First.

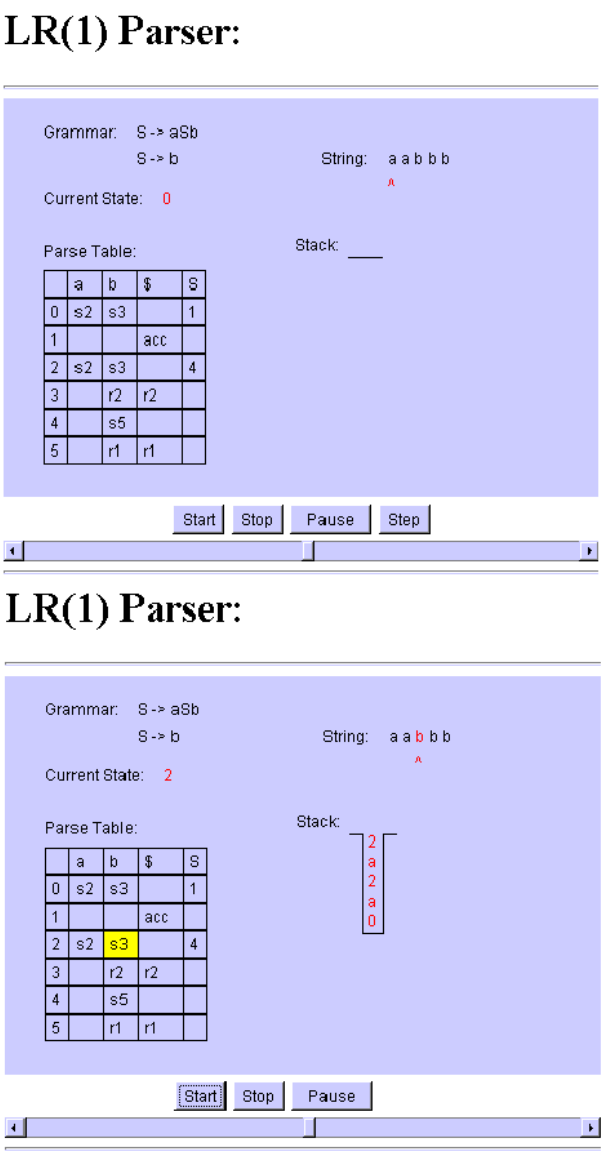

Figura 2.14: Parser LR(1)

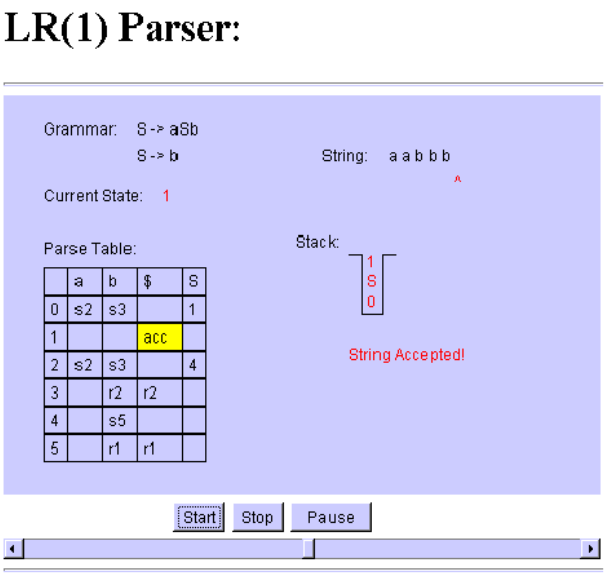

Figura 2.15: Parser LR(1) (continuação)

2.5.5 Tipo IV

O sistema Alma surge para preencher este espaço, visto ser um sistema multi-algoritmo e multi-linguagem. Este sistema não altera o código fonte e gera automaticamente a animação com base na representação interna dos programas.

2.6 Animação em Geradores de Compiladores

Estes sistemas caracterizam-se por serem destinados essencialmente à construção de compiladores integrados em ambientes de programação mas, por recorrerem à descrição formal de linguagens e à representação interna de programas, geram algumas visualizações sobre o processo de geração do compilador ou permitem a definição de processos de animação.

2.6.1 O sistema LRC

LRC ([Sar99],[KS98]) é um sistema de construção de ambientes de programação, sendo essa construção baseada na especificação formal da linguagem na qual se pretende programar. A especificação léxica, sintáctica e semântica da linguagem é feita através de gramáticas de atributos de ordem superior.

Este sistema aceita como entrada uma gramática de atributos e gera calculadores de atributos puramente funcionais em C e em Haskell.

A arquitectura do sistema LRC é constituída por três partes: processador de gramáticas de atributos de ordem superior; um gerador de funções de visita; e um sistema de *runtime* que contem um calculador incremental e um visualizador. O processador gera, a partir de uma gramática de atributos de ordem superior, especificações para os geradores léxico e sintáctico, um conjunto de funções para o calculador de atributos de análise semântica e uma descrição abstracta da gramática de atributos. Esta descrição abstracta é feita numa linguagem intermédia, independente da linguagem de entrada.

O gerador de funções de visita é o *back-end* do sistema que processa a gramática de atributos abstracta e gera calculadores de atributos puramente funcionais.

O sistema de *runtime* efectua cálculo incremental e constrói as visualizações. As ferramentas geradas pelo sistema contêm interfaces gráficas avançadas. Estas interfaces são descritas dentro do formalismo das gramáticas de atributos. Durante o cálculo de atributos, um dos atributos a ser sintetizado é a descrição abstracta da interface. O visualizador também é incremental: quando a árvore de sintaxe sofre uma alteração, a interface é novamente calculada e o visualizador actualiza as partes alteradas.

Na medida em que o sistema LRC permite especificar ambientes de programação para uma dada linguagem, tem acesso à parte semântica e ao cálculo de atributos efectuado durante a execução de um programa. Permite também especificar o *layout*, ou seja, o formato em que são apresentados os resultados.

Este sistema permite, então, com relativa facilidade, aceder a valores intermédios das variáveis de um programa e apresentá-los de forma elucidativa (por exemplo, sob a forma de animação do programa). A animação pode ser construída anotando a gramática abstracta com funções de desenho. Nessas situações, o sistema LRC pode ser visto como um sistema de animação do tipo IV (multi-algoritmo e multi-linguagem), no entanto, o sistema não está vocacionado para a construção de animações de programas e poderá não ser fácil gerar animações de programas mais complexos.

2.6.2 O sistema CENTAUR

O sistema CENTAUR [Ber91] foi criado para desenvolvimento de ambientes de programação. A construção desses ambientes baseia-se na descrição formal axiomática (lógica natural) das linguagens de programação para as quais os ambientes vão ser gerados. As descrições formais focam aspectos sintác-

ticos, como gramáticas independentes de contexto e descrições de *layout's*, e aspectos semânticos como *disciplina de tipos* e *semântica operacional* ou tradução para outras linguagens.
Os autores apresentam ainda o conceito de *subject tracking* que permite associar a execução a posições do programa. Cada instrução aparece associada a uma posição e define uma ocorrência.
As manipulações formais que integram computações de ocorrências são sistemáticas e automatizadas. Os autores acreditam que esta técnica poderá ser usada para animar programas ou para colocar *break-points* no interpretador gerado a partir destas especificações formais.

2.6.3 O ambiente SmartTools

SmartTools [AP02] permite o desenvolvimento rápido de ambientes de programação: produz um editor estruturado e ferramentas de visualização. Este sistema é um sucessor do Centaur, usa os mesmos conceitos básicos como a especificação baseada na árvore de sintaxe abstracta.
De um formalismo AST (*Abstract Syntax Tree*) que descreve uma dada linguagem, SmartTools gera automaticamente um editor estruturado para essa linguagem. Assim, o utilizador pode editar qualquer ficheiro nessa linguagem usando esse editor e dispõe também de algumas ferramentas genéricas de visualização.
A especificação do utilizador não tem que seguir a sintaxe AST do SmartTools. Em alternativa, a especificação pode ser feita através de DTD's, seguindo a norma XML, porque o SmartTools inclui um conversor DTD/AST. Para além do editor, poderão ser gerados um *parser* e um *pretty-printer*. Para tal, o utilizador deve indicar mais informação sobre atributos extra que são necessários.
Para o SmartTools, cada árvore de sintaxe abstracta representa um documento a partir do qual se pode construir e mapear uma ou mais vistas. Por exemplo, para um texto pode ser produzido um texto formatado (pelo *pretty-printer*) ou uma representação gráfica da árvore ou informação sobre algum nodo seleccionado. Para tal, são usadas regras de transformação (da árvore) definidas na linguagem Xpp. Cada regra é um par que faz corresponder a cada padrão (nodo com restrições aos seus filhos ou seus atributos) o código da formatação desejada. O utilizador deve então definir na notação Xpp as funções correspondentes aos operadores da sua linguagem. Neste sentido, se o utilizador associar funções de desenho, poderá criar visualizações de programas escritos nessa linguagem.
O SmartTools pode também gerar ferramentas semânticas (para transformação dos textos fonte), mas à custa de *visitor patterns* que deverão ser instanciados pelo utilizador.

Na implementação do SmartTools foi usado o Java Swing API, o que permitiu obter uma interface gráfica de boa qualidade e de fácil de configuração.

2.6.4 O sistema LISA

O sistema LISA é uma ferramenta de implementação automática de linguagens. Esta ferramenta [MLAZ00] produz compiladores e interpretadores em Java, a partir de uma especificação formal da linguagem. O ambiente gerado inclui um editor, um compilador/interpretador e outras ferramentas gráficas. Este ambiente integrado permite especificar e gerar analisadores, compilar e executar os programas fonte.
Na análise léxica e sintáctica são usadas expressões regulares e a notação BNF, respectivamente. A semântica da linguagem é definida através de gramática de atributos com possibilidade de recorrer a herança múltipla, o que permite definir uma linguagem de forma incremental ou reutilizar especificações de outras linguagens. Os analisadores sintáctico e semântico podem ser de vários tipos: LL, SLR, LALR e LR (os sintácticos) e *tree-walk*, *parallel*, *L-attribute* e *Katayama* (os semânticos).
A construção deste novo sistema de geração de compiladores justifica-se pela tentativa de melhorar a implementação desses compiladores usando técnicas de desenvolvimento incremental, visualização de análises (léxica, sintáctica e semântica) e aumento da portabilidade. O sistema LISA foi desenvolvido usando tecnologia orientada aos objectos(Java) e corre em MS Windows. Os utilizadores do compilador gerado têm a possibilidade de visualizar o trabalho dos analisadores léxico, sintáctico e semântico, através dos autómatos de estados finitos, diagramas de sintaxe e diagramas semânticos.
O sistema constrói automaticamente visualizações de estruturas de dados (estruturas internas do compilador gerado) e uma animação do processo de cálculo dos atributos. No entanto, a animação e as visualizações baseiam-se na meta-linguagem usada para especificar as gramáticas sendo, portanto, dependentes dessa linguagem. Assim, relativamente à construção destas animações o sistema pode apenas ser comparado com os sistemas de animação do tipo III (multi-programa mas mono-linguagem).

2.7 Análise dos sistemas de animação do Tipo III

Em 1997, Stasko propôs uma grelha para classificação/comparação de sistemas de animação. Naturalmente surgiu a preocupação de a relacionar com o sistema de classificação aqui proposto (que foi apresentado em 2.4 e usado em 2.5 e 2.6). Concluiu-se então que não são alternativos mas complementares, embora a de Stasko seja de aplicação mais reduzida.

Para completar este estudo e ilustrar a conclusão referida, os sistemas de animação classificados como sendo do tipo III (ver subsecção 2.5.4), foram sujeitos a uma classificação de acordo com os critérios estabelecidos por J. Stasko em [SDBP97].

2.7.1 Parâmetros de análise

Foram definidos, por Stasko, seis parâmetros principais que deverão ser usados na classificação de um sistema de visualização de software:

- **área de abrangência** (scope);
- **conteúdo**;
- **forma**;
- **método**;
- **interacção**;
- **aplicabilidade**.

Quanto à **área de abrangência** e em **termos gerais**, é necessário verificar qual o hardware e qual sistema operativo necessários; qual a linguagem fonte a utilizar; se o sistema pode ou não visualizar aspectos concorrentes; se existem restrições quanto aos programas a serem visualizados; se o sistema tem ou não algum tipo de programa o qual é especialista em visualizar. Em termos de espectro de **visualização**, o sistema poderá abranger variados aspectos e perspectivas dos programas fonte.
Quanto ao **conteúdo** e a nível de **programa**, poderão ser visualizados **código** ou **dados**. Quanto ao **código** poderá ser avaliado até que ponto são visualizadas as instruções do programa; e se é ou não visualizado o fluxo de controlo; quais os tipos de visualização de código disponíveis : código fonte *pretty printed*; diagramas estruturados; árvore de chamadas a subprogramas. Quanto à visualização de **dados** deve ser verificado o grau de detalhe da visualização das estruturas, e do fluxo de dados; qual o tipo de visualização usado: desenho das estruturas mostrando os seus conteúdos; ou diagramas de fluxo de dados.
Quanto ao **conteúdo** e a nível de **algoritmo**, podem ser visualizados dados ou instruções (fluxo de dados ou fluxo de controlo respectivamente). Quanto ao **conteúdo** e a nível de **fidelidade** é necessário verificar se as visualizações mostram o comportamento completo da máquina virtual subjacente. Ainda quanto ao

conteúdo mas relativamente ao **momento de recolha de dados**, alguns sistemas não conseguem obter visualizações com dados actualizados porque não têm acesso a esses valores em tempo de execução. É necessário definir o critério de controlo temporal entre o programa e a respectiva visualização. A visualização é produzida a partir de dados guardados durante uma execução prévia ou é produzida à medida que o programa é executado. Se a visualização é baseada em informação recolhida num determinado ponto de execução e é gerada uma visualização estática então o *mapping* é *static to static*. Se a visualização é animada, o *mapping* é *static to dynamic* (não há exemplos). Se a visualização recolhe informação ao longo da execução e produz uma visualização única então é *dynamic to static*. Se a visualização produzida é animada então o *mapping* é *dynamic to dynamic*.

Quanto à **forma** das visualizações geradas, pode ser avaliado o **estilo da apresentação**: o vocabulário gráfico (cor, dimensões); a animação e o som. Pode também ser avaliada a **granularidade**, ou seja, se as visualizações são mais ou menos detalhadas ou se se pode ocultar as partes menos importantes. O sistema pode ou não permitir a **sincronização de programas** (usada, por exemplo, para comparar a velocidade de dois programas) e pode ou não permitir **visões múltiplas** de dados e código, ou seja, visões simultâneas de diversas informações sobre o programa.

Quanto ao **método** utilizado para gerar as visualizações, pode-se avaliar o **estilo de especificação** da visualização e a **técnica de conexão**. O **estilo de especificação** pode ser feito à mão (criar os desenhos das visualizações de raiz usando uma determinada linguagem de programação) ou com base em bibliotecas de visualizações de forma mais ou menos automática. Pode ser avaliada a qualidade de uma visualização automática para medir a **inteligência** do sistema. Pode também ser medido o **grau de controlo** que o utilizador tem sobre a visualização. Quanto à **técnica de conexão**, alguns sistemas recorrem a: adicionar instruções de visualização; à anotação automática; a adicionar primitivas de exploração às estruturas de dados sem que o código fonte seja alterado; ou a usar um dispositivo que osculta o barramento e mostra um relatório vivo dos comandos que estão a ser executados. Não sendo, nestes casos, a geração da visualização totalmente automática irá ser necessário mais ou menos quantidade de conhecimento do código do programa. Poderá também ser avaliado o quanto emparelhado está o sistema de visualização com o código. Alguns sistemas requerem que o programa a ser visualizado seja escrito no ambiente do próprio sistema.

Quanto à **interacção** de um sistema de visualização, pode ser avaliado o **estilo de interacção**, ou seja, que método usa o utilizador para dar instruções ao sistema (botões, menus, instruções de linha de comando); e também a **navegação**, ou seja, até que ponto, o sistema suporta a navegação pela visualização.

É um aspecto importante para programas muito grandes e a navegação deve ser conseguida com base na mudança de resolução, escala, compressão, selecção e abstracção. A navegação deve permitir suprimir detalhes e controlar aspectos temporais (direcção reversível, velocidade de execução). O sistema deve ser também avaliado quanto à facilidade de ***scripting***, ou seja, facilidade de guardar um conjunto de interacções sobre uma visualização (para o caso de demonstrações).

Quanto a **situações de aplicação**, deve ser visto qual o **objectivo** do sistema (ser usado em salas de aula ou em demonstrações de algoritmos complexos ou em processos de debugging). O sistema deve ser avaliado pela forma como comunica a informação ao utilizador. Deve também ser verificada a **adequação** do sistema, ou seja, deve ser avaliada a rapidez com que as metáforas visuais inspiram a compreensão e o entendimento dos programas. Deve também ser feita uma **avaliação empírica** do sistema e deve também ser avaliado o seu uso, ou seja, o período de tempo que está em uso quer a nível da educação quer ao nível da indústria.

Relativamente à área de abrangência, a maior parte dos sistemas trabalham apenas com pequenos programas e programas escritos em linguagens específicas, portanto a área de abrangência limita-se a uma determinada linguagem.

Quanto ao conteúdo dos sistemas varia entre programas e algoritmos. Normalmente, os alunos deverão usar sistemas de visualização de algoritmos para não se preocuparem com detalhes de implementação. No entanto, para programadores seria desejável um sistema que permitisse transições fáceis entre programa e algoritmo. Pouco trabalho tem sido feito para ser possível mostrar o fluxo de controlo e o fluxo de dados em *run-time*.

Quanto à forma das visualizações geradas, Stasko é de opinião que deveria ser mais usada a cor, o som, e os *output's* multi-dimensionais. É necessário explorar o aspecto da granularidade para ultrapassar problemas de escalonamento e embora já tenham sido usadas visões múltiplas, é necessário melhorar as interfaces que as controlam.

Quanto ao métodos de especificação a tendência é para ser cada vez mais automatizados e não devem requerer que o programador tenha que entender o código do programa para poder produzir a visualização. Os novos sistemas deverão permitir navegação avançada pelos grandes programas e era desejável que se criassem protótipos que usem tecnologias virtuais de navegação. Deverão também ser incluídas algumas facilidades de *scripting* e deverão ser feitas mais avaliações sobre a aplicabilidade dos sistemas.

2.7.2 Estudo comparativo dos sistemas de animação do tipo III e do sistema proposto

Os sistemas de animação do tipo III que vão ser avaliados serão : VIP, JELIOT, JCAT, ZSTEP, LENS. Por último, será classificado o resultado de implementar o sistema Alma, proposto nesta tese.

NOME	ÁREA DE ABRANGÊNCIA		
	TERMOS GERAIS		
	SO	Linguagem	Aplicação
VIP	MS-DOS	Pseudo-Pascal	peq progs
JELIOT	XWindows	Java	progs java
JCAT	Windows	Java	progs java
ZSTEP	Macintosh	Pseudo_Lisp	peq progs
LENS	Unix/XWin	C	progs C
ALMA	Windows/Linux	qq	qq prog

Tabela 2.2: Classificação de alguns sistemas de animação quanto à área de abrangência

Para os sistemas em estudo, a tabela 2.2 indica os sistemas operativos que os suportam, a linguagem fonte a utilizar (no caso do Alma pretende-se que seja uma qualquer) e o tipo de programas a que se aplicam.

NOME	CONTEÚDO					
	PROGRAMA		ALGORITMO		Fidelidade	Momento de recolha
	Código	Dados	Instruções	Dados		
VIP	não	sim	não	não	boa	simultâneo
JELIOT	não	sim	não	não	boa	simultâneo
JCAT	não	sim	não	não	boa	simultâneo
ZSTEP	sim	sim	não	não	boa	simultâneo
LENS	não	sim	não	não	razoável	simultâneo
ALMA	sim	sim	sim	sim	boa	simultâneo

Tabela 2.3: Classificação de alguns sistemas de animação quanto ao conteúdo das visualizações

Relativamente ao conteúdo das visualizações é necessário esclarecer algumas noções. Visualização de programas é diferente de visualização de algoritmos: visualizar um algoritmo é obter uma representação

mais abstracta e mais alto nível de um conjunto de operações; visualizar código de um programa é aprender algo sobre uma determinada implementação do algoritmo. Alguns sistemas são suficientemente flexíveis para gerar os dois tipos de visualização.

Os sistemas em estudo são dependentes da linguagem fonte (excepto o sistema Alma) e a visualização gerada está inteiramente relacionada com cada uma das instruções do programa fonte. Pretende-se, no sistema Alma, não só permitir o uso de linguagens fonte mais algorítmicas mas também associar desenhos mais abstractos aos conceitos envolvidos nesses algoritmos.

Quer a visualização de programas quer a de algoritmos divide-se ainda em visualizar código ou dados. A visualização de código pode ser feita através da apresentação do código fonte *pretty-printed*, diagramas estruturais, árvores de invocação, etc. Neste sentido nenhum dos sistemas em estudo tem por objectivo produzir este tipo de animações (visualização explícita do código fonte) embora o sistema ZStep produza um historial das instruções executadas. Normalmente, é apenas gerada a visualização de dados e a alteração dos seus valores ao longo da execução das instruções do programa. Isto faz com que de uma forma mais ou menos perceptível as instruções do programa sejam também facilmente identificadas no decorrer da animação. No caso do sistema Alma há uma representação visual do código do programa e de cada uma das variáveis envolvidas (e seus valores) em cada instrução.

Os sistemas com propósitos pedagógicos têm alguma tendência a ignorar aspectos mais detalhados do programa para que a explicação visual gerada não se torne demasiado complexa. Assim, a animação perde alguma fidelidade ao programa fonte e completude.

Todos os sistemas em estudo geram animações em *run-time*, portanto, o momento de recolha coincide com o momento de geração da animação. Uma animação gerada em *compile-time* nunca poderia ter acesso às diversas instâncias das variáveis ao longo do programa.

Relativamente à forma da visualização foram avaliados parâmetros relativos: ao estilo (textual, visual ou ambos);ao detalhe da visualização; ao facto de permitir a visualização de parte do programa ou visualizações múltiplas e sincronizadas de estruturas de dados.

Relativamente ao método usado para construir as animações foi verificado o tipo de especificação usada, a qualidade da geração automática de visualizações (inteligência) e o grau de controlo do que vai ser gerado (introdução de valores para os parâmetros de animação). O sistema Alma poderá funcionar em vários modos: num modo mais básico onde o utilizador submete o seu programa e obtém a animação sem qualquer indicação adicional e um modo onde poderá mudar o aspecto da animação, mudar o nível

NOME	FORMA				
	Estilo	Detalhe	Partes	Sincronização	Visões Múltiplas
VIP	textual	algum	não	não	sim
JELIOT	visual	bastante	não	sim	sim
JCAT	visual	bastante	sim	sim	sim
ZSTEP	tex/visual	bastante	sim	não	não
LENS	visual	algum	não	não	não
ALMA	visual	bastante	sim	não	não

Tabela 2.4: Classificação de alguns sistemas de animação quanto à forma das visualizações

NOME	MÉTODO		
	ESTILO DE ESPECIFICAÇÃO		
	Especificação	Inteligência	Grau de Controlo
VIP	ling. específica	alta	nenhum
JELIOT	tipos especiais	alta	bastante
JCAT	anotações	alta	total
ZSTEP	ling. específica	alta	pouco
LENS	assoc. desenhos	razoável	total
ALMA	não se aplica	alta	variável

Tabela 2.5: Classificação de alguns sistemas de animação quanto ao método de especificação das visualizações (Estilo de Especificação)

de detalhe ou até mesmo definir nova semântica.

O parâmetro apresentado na tabela 2.6 avalia o tipo de conexão entre o código fonte e a visualização gerada.

A nível de interacção com o utilizador diversos parâmetros podem ser considerados: capacidade de expansão ou supressão de informação; controlo de aspectos temporais da execução; capacidade de inversão da direcção temporal da visualização; controlo da velocidade de execução. Neste caso, apenas foi avaliado o estilo de interacção indicando o tipo de editor utilizado, a qualidade de navegação e a existência da técnica de *scripting* (capacidade de gravar diversos passos de interacção para serem usados em futuras demonstrações).

NOME	MÉTODO
	TÉCNICA DE CONEXÃO
VIP	uso de linguagem específica com visualização pré-definida
JELIOT	anotação automática do código (tipos de dados especiais)
JCAT	anotação do código com primitivas de desenho
ZSTEP	uso de linguagem específica com visualização pré-definida
LENS	conexão manual entre desenhos e instruções
ALMA	geração automática da visualização

Tabela 2.6: Classificação de alguns sistemas de animação quanto ao método de especificação das visualizações (Técnicas de Conexão)

NOME	INTERACÇÃO		
	Estilo	Navegação	Scripting
VIP	editor est.	não	não
JELIOT	editor text/vis	alguma	não
JCAT	editor textual	bastante boa	não
ZSTEP	editor textual	muito boa	não
LENS	editor text/vis	pouca	não
ALMA	editor textual	bastante boa	não

Tabela 2.7: Classificação de alguns sistemas de animação quanto à interacção do sistema

Quando um sistema é avaliado em termos de aplicabilidade é necessário identificar a situação em que é usado, se é usado com eficiência e se foi alguma vez sujeito a algum tipo de avaliação. A tabela 2.8 é uma tentativa de análise dos sistemas em estudo quanto a estes parâmetros.

As tabelas de classificação aqui apresentadas carecem de uma actualização continua porque surgem constantemente novas técnicas de especificação das visualizações, cada sistema têm o seu objectivo específico e surgem novos formatos de visualização, novas interfaces, etc. No entanto, há características que são inspeccionadas sempre em todos os sistemas relacionadas com os resultados que produzem: visões múltiplas/monolíticas; história estática/dinâmica; transições continuas/discretas; escolha dos dados de entrada (quantidade e tipo); distinção entre os vários estados de um algoritmo; distinção entre a

NOME	SITUAÇÕES DE APLICAÇÃO		
	Uso	Eficiência	Avaliação
VIP	aulas	médio	sim
JELIOT	aulas	bom	Alguma
JCAT	aulas	bom	Alguma
ZSTEP	prog/aulas	bom	sim
LENS	prog/aulas	médio	?
ALMA	aulas	?	?

Tabela 2.8: Classificação de alguns sistemas de animação quanto a situações de aplicação

actividade actual das restantes (dentro do algoritmo).

Mais recentemente, foram criados sistemas que usam tipos de dados especiais, anotação automática ou geração automática de animações, portanto, seria útil incluir neste tipo de classificação parâmetros como: grau de automatização da construção de visualizações; quantidade e tipos de vistas geradas para um mesmo programa; grau de independência entre linguagem fonte e sistema de animação; grau de alteração do texto fonte requerido pelo processo de animação.

Em termos de classificações de sistemas de visualização, é de notar a existência de uma página na *web* [Bur01] onde os organizadores das conferências sobre linguagens visuais e afins como *visualização de software*, mantêm referências aos inúmeros trabalhos desta área. As referências relativas ao tema *visualização de software* surgem classificadas segundo os seguintes parâmetros: técnica (como é feita a interacção com o utilizador; como é gerado o output, ou seja, como é criada a visualização); aplicações (onde pode ser aplicado o trabalho ou sistema); eficiência da visualização gerada; domínio da visualização (o que é visto, tipo de visualização, qual o paradigma abrangido).

2.8 Conclusão sobre o actual estado da arte

Relativamente aos sistemas de animação encontrados foram apresentadas as várias técnicas usadas: uso de bibliotecas de funções de visualização; manipulação directa da animação; anotação de algoritmos; tipos de dados auto-animados; linguagens específicas de animação; visualizações declarativas; anotação da semântica de programas. Foi também apresentada uma proposta de classificação desses sistemas. Os tipos de sistemas foram identificados de 0 a 4 e foram apresentados vários sistemas em cada um

desses tipos. Os sistemas do tipo 0, embora não sejam sistemas de animação, permitem a construção das mesmas com objectivo de simular funcionalidades e comportamentos. Foram classificados como sendo do tipo I todas as animações já criadas para algoritmos específicos e que não permitem qualquer tipo de interacção com o utilizador. São exemplo todas as referências de [BV99] onde se faz uma listagem completa de animações que se podem encontrar em paginas da Internet. Essa colecção está dividida em vários tipos: algoritmos de ordenação (*Bubble Sort*, *Merge Sort*, *Quick Sort*, ...); algoritmos sobre árvores (árvores B, procura binária em árvores, procura *Depth First*); algoritmos geométricos (Fecho Convexo,...); algoritmos de grafos (grafos de matrizes de adjacência, procura em grafos, cálculo do caminho mais curto, ...); algoritmos paralelos (ordenação paralela, Fecho Convexo paralelo); algoritmos para manuseamento de estruturas de dados (arrays, tabelas de hashing, queues, stacks); outros algoritmos (Parser LR(1), multiplicação de matrizes, jogo do ping pong, torres de Hanoi, ...).

Classifica-se como tipo II as animações que permitem alguma interacção. Os sistemas do tipo III permitem a visualização de programas obrigando a que estes estejam escritos numa determinada linguagem, e os de tipo IV são sistemas de animação multi-algoritmo e multi-linguagem que geram automaticamente animações de um programa a partir do seu texto.

Neste capítulo foram também introduzidos e relacionados alguns sistemas de classificação propostos por outros autores. Foram apresentadas grelhas de comparação de alguns dos sistemas já referidos, usando um critério proposto por Stasko.

O sistema Alma pretende ser um novo sistema de visualização de programas baseado na representação interna dos mesmos. Isto permite evitar qualquer tipo de anotação de código e permite receber programas escritos em diversas linguagens. A animação será gerada automaticamente sendo apresentado num editor visual todos os seus passos, permitindo que o utilizador navegue na animação quantas vezes quiser e à velocidade que quiser.

Capítulo 3

O sistema Alma

Neste capítulo serão apenas apresentados os princípios e objectivos subjacentes à concepção do sistema Alma bem como as suas finalidades, na linha do que foi apresentado em [VH99] e em [VH00].

3.1 Apresentação do sistema

Como já foi dito, este trabalho de doutoramento tem como objectivo propor uma solução para o preenchimento da lacuna detectada no capítulo anterior (revisão sobre o estado da arte): um sistema do tipo IV. Um sistema de animação deste tipo, tal como aí foi mencionado, é um sistema de animação multi-algoritmo e multi-linguagem que gera automaticamente animações dos programas fonte que lhe forem submetidos. O sistema Alma pretende cumprir esses objectivos: aqui discutem-se os princípios, sendo a sua arquitectura apresentada nos capítulos seguintes.

O sistema Alma terá que ser capaz de analisar o programa fonte, que se pretende *animar* —permitindo, *visualizar o fluxo de controlo e/ou dos dados durante uma simulação da sua execução*. Este sistema pretende demonstrar que é possível fazer a animação dos algoritmos subjacentes aos programas de uma forma sistemática, isto é, sem depender nem de um programa (ou classe de programas) específico, nem tão pouco de uma linguagem de programação determinada. Para tal, recorre-se a métodos, técnicas e ferramentas tradicionalmente usadas para reconhecer, representar e manipular o significado dos programas, no contexto do desenvolvimento formal e automático de compiladores.

O sistema Alma, tal como todos os sistemas de animação de programas, tem como objectivo facilitar a compreensão do programa e do algoritmo subjacente. Para um iniciante na programação, um sistema

deste tipo ajuda a perceber as tarefas executadas e as alterações que provocam nos valores das variáveis; detectar erros; verificar se o programa cumpre os objectivos para que foi feito.

O sistema Alma quando aplicado a uma linguagem de domínio específico produz um conjunto de visualizações (animação) que descrevem comportamentos nas mais diversas áreas de aplicação (sem ser forçosamente algo relacionado com operações, operandos, instruções ou outros conceitos relacionados com as linguagens de programação). Mesmo que a linguagem fonte seja uma linguagem só de especificação, mais declarativa, a sua visualização (embora estática) tem também utilidade.

A representação visual produzida contem informação sobre instruções e dados. O utilizador tem a percepção do evoluir da execução do programa (desenho das instruções do programa) e das alterações dos valores das variáveis (desenho de cada variável incorporado no desenho da instrução). Este resultado não tem o mesmo objectivo dos resultados produzidos por um debugger: não se pretende corrigir erros mas apenas obter uma visão diferente do programa fonte. Naturalmente, fará mais sentido usar um sistema de animação como o Alma em pequenos programas onde o intuito é apenas explicar conceitos envolvidos na linguagem fonte que o utilizador pretende estudar.

O sistema Alma, para além de poder ser aplicado ao ensino da programação, pode também ser útil a outras matérias que recorrem a programas de computador, como é o caso da matemática. Por exemplo, a visualização de um programa em Fortran para cálculo de uma série de Fourrier poderá ajudar a entender não o programa em si mas cálculo efectuado.

O texto fonte de um sistema da animação poderá, como já foi dito, representar conceitos muito diversos, desde um simples programa em Pascal para calcular uma expressão, passando por linguagens mais declarativas ou até um documento estruturado do qual se pretende também obter uma representação visual. Outras aplicações poderão surgir em áreas completamente distintas. Por exemplo, se for construída uma representação visual das respostas de escolha múltipla de um teste, essa imagem poderá ser comparada com a visualização das respostas correctas, para se obter mais facilmente a sua classificação.

Assim, como aplicações possíveis do Alma, destacamos: a animação de algoritmos, como apoio ao ensino da programação e como instrumento da didáctica da matemática (onde o principal uso, poderá ser na explicação visual dos princípios de cálculo descritos pelo algoritmo em análise); a análise de respostas, para apoio à correcção de provas de avaliação; a interpretação (visual) de documentos anotados.

3.2 Princípio, evolução e objectivos do sistema Alma

A concepção do sistema Alma teve como objectivo fazer com que as suas principais características fossem: a independência relativamente ao programa fonte e à linguagem de programação; e ao mesmo tempo, um sistema versátil permitindo adaptá-lo às necessidades do utilizador.

Quando se pretende visualizar o fluxo de controlo ou de dados podemos considerar que o programa é um texto e, como tal, é estático. Assim, a visualização de um programa não é mais do que uma representação visual do código. No entanto, a animação de um programa pretende sobretudo mostrar o comportamento das variáveis e seus valores, o que é conseguido à custa de várias visualizações. Neste sentido, teríamos à partida duas grandes escolhas: fazê-lo durante a execução do código (debugging); ou simular a execução num outro ambiente.

O executável do programa, por si só, em nada ajuda a entender o funcionamento do programa. Por outro lado, a execução em tempo real de um programa é demasiadamente rápida para haver percepção do fluxo de controlo e até da evolução dos valores das variáveis. Tal análise só é possível com a ajuda de um debugger, mas para isso o compilador tem de ser prevenido, de modo a gerar código adicional apropriado.

Face ao exposto, parece ser uma boa solução recorrer a simulações da semântica dinâmica do programa num ambiente que permita estudar o seu comportamento.

Uma segunda escolha que tem de ser feita de seguida é se se deve, ou não, incluir no programa fonte instruções e/ou marcas especiais a indicar que blocos de código e que variáveis é que se pretende estudar durante a animação. Em caso de escolha afirmativa, será necessário definir a linguagem de animação que o programador deve conhecer para incluir nos seus programas.

Aqui a escolha foi, claramente, pela negativa —não pretendemos que haja uma linguagem de marcação de animação extra, pois não desejamos que os programas a animar tenham de ser previamente alterados. Tal decisão implica que o ambiente de animação tenha capacidade para: analisar o programa fonte — identificando o seu fluxo de controlo e as suas variáveis; permitir ao programador controlar, na hora da animação, os blocos e variáveis a estudar.

A implementação de um pequeno protótipo para animar um programa Java (programa que inspecciona de x em x tempo os valores de um array e actualiza o desenho, demonstrou que existe, em certas linguagens, uma grande dificuldade em aceder a certos tipos de informações sobre o programa fonte sem que este seja modificado. Conclui-se no entanto, que um sistema que obtenha acesso a nomes e endereços

de variáveis (tabela de identificadores) e a certas informações sobre controlo de fluxo de um programa (diagramas de fluxo de dados e de controlo), usando métodos mais alto nível (análise léxica, sintáctica e semântica), conseguirá animar programas escritos em diferentes linguagens desde que conheça a respectiva gramática.
Optou-se, assim e no domínio da animação de linguagens textuais, por uma solução mais genérica, descrita nas secções seguintes.

Em conclusão, o sistema Alma tem por objectivo animar programas criando visualizações de simulações desses programas. Visualizar um programa é distinto de visualizar uma simulação de execução desse programa. Visualizar um programa pode ser simplesmente uma tradução do programa textual para um programa visual. Visualizar uma simulação da execução de um programa significa mostrar as operações que vão sendo efectuadas e que influência têm nos valores das variáveis.
Visualizar uma simulação da execução de um programa não se faz através de um só desenho (uma representação visual) mas de uma sequência de desenhos que vão mostrando a evolução de instruções e variáveis. A isso chamamos animação.
Com a construção do sistema Alma pretende-se, então, cumprir os seguintes requisitos:

- construir um ambiente integrado e fácil de usar
- evitar a necessidade de alterar o código fonte
- permitir a selecção de diferentes vistas do mesmo programa
- criar um sistema o mais genérico possível de maneira a ser usado por várias linguagens fonte

3.3 O caracter genérico do sistema

Como tem vindo a ser dito, a filosofia de construção que escolhemos para o sistema Alma permite que ele seja adaptável a diferentes linguagens de programação e até a diferentes famílias de textos-fonte. A cada texto de entrada corresponde uma gramática que deverá ser conhecida pelo sistema. O sistema, a partir do momento que consiga reconhecer as frases da linguagem definida por essa gramática, consegue construir uma representação interna de cada texto disponibilizando as informações necessárias sobre o conteúdo dos programas para proceder à sua animação. Na figura 3.1 fica visível a entrada e a saída do sistema de animação que se pretende conceber: tendo como entrada programas escritos em várias lin-

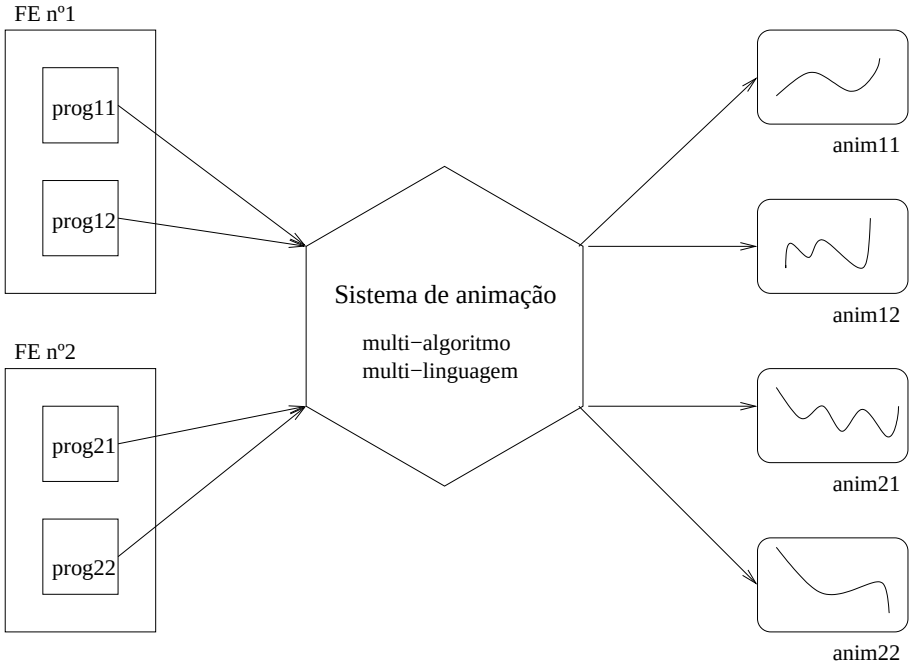

Figura 3.1: Concepção do Sistema Alma

guagens (eventualmente pertencentes a paradigmas diferentes), gera a respectiva animação. Esta parte de adaptação do Alma a diferentes situações, embora não seja tarefa a ser executada pelo utilizador final, é um trabalho que pode ser realizado sistemática e facilmente pelo implementador (se este for especialista em desenvolver processadores de linguagens). Utiliza-se o mesmo *back-end*, ou seja, a animação é gerada sempre da mesma forma independentemente da linguagem, o qual trabalha sempre sobre a mesma representação interna (que terá de ser produzida pelos diferentes *front-end*).

A generalidade do sistema opõe-se à expressividade do mesmo, ou seja, se usarmos o mesmo tipo de visualizações em diferentes linguagens, para tentar preservar a generalidade do sistema, perderemos em expressividade se não estivermos a utilizar a visualização adequada.

As interfaces do sistema devem permitir que o utilizador interactue facilmente com o sistema fornecendo entre outras ferramentas um editor de texto. Ora a gramática que se define para gerar o reconhecedor (que constitui o *front-end*) pode ser, igualmente, usada para gerar um Editor estruturado Dirigido pela Sintaxe. Assim, a interacção com o animador é francamente melhorada, não só em funcionalidade, como também em facilidade de utilização.

O sistema deverá permitir também escolher alguns tipos de visualizações a obter. O tipo de visualização depende do objectivo que se pretende atingir ao fazer a análise de um programa. Sabendo que um pro-

grama tem sempre subjacente determinadas estruturas, as visualizações irão basear-se em representações visuais das estruturas de dados e do controlo de fluxo do programa, que de alguma forma contribuirão para o objectivo da análise que se quer efectuar.

3.4 Finalidades do Sistema

Tal como já foi abordado na secção 3.1 deste capítulo, pode-se considerar como possíveis aplicações do sistema Alma as seguintes:

- ensino da programação através da análise de programas / algoritmos,
- ensino da matemática através da visualização detalhada do algoritmo de cálculo contido no programa,
- análise de documentos, vendo um documento como sendo um programa escrito segundo um DTD
- análise de respostas a problemas, visualizando a solução gerada pelos programadores a serem avaliados.

A finalidade do sistema proposto é analisar um texto de entrada (que poderá ser um programa clássico, um algoritmo, um documento, etc.) e extrair a informação considerada importante que, depois de visualizada de uma forma clara e sugestiva, permite entender melhor o significado do texto fonte.
Se texto de entrada for um programa escrito numa linguagem imperativa, a informação a extrair terá que permitir compreender o fluxo de controlo e o comportamento das estruturas de dados.
Se, por outro lado, a entrada do sistema consistir nalgum algoritmo (escrito numa qualquer notação) que descreva um determinado tipo de cálculo numérico, poderá ser também interpretado como sendo um programa, escrito em Fortran ou outra linguagem apropriada para o efeito. Neste caso, a informação deverá ser visualizada de forma a que o utilizador perceba o algoritmo subjacente ao programa porque este irá permitir compreender os passos necessários para efectuar o cálculo numérico em si.
Se a entrada for um documento marcado, este deverá ser encarado também como um programa escrito na linguagem de anotação gerada por um DTD. Neste caso, o sistema também pode reconhecer a estrutura do documento —de acordo com o seu tipo, definido pelo DTD subjacente a esse documento— e, então, mostrar a sua forma. Assim, será possível evidenciar determinado tipo de anotações, segundo a pretensão do utilizador, tal como se faz nos programas relativamente às estruturas de dados; poder-se-á ilustrar os

locais aonde certas partes do documento são referidas e evidenciar relações entre determinadas marcas. Como é evidente apenas é gerada uma visualização.

Assim, o sistema poderá não só ser aplicado em situações de visualização de abstracções de programas, mas também em situações de verificação de textos onde também é necessário efectuar uma análise profunda à estrutura e ao conteúdo do texto fonte.

Este é o caso da última aplicação, referida acima, em que se antevê a hipótese de usar o sistema de animação para interpretar visualmente resultados produzidos por programas em resposta a problemas.

3.5 Características da interface para um sistema deste tipo

Tendo sido projectado o sistema Alma para ser usado como ajuda a tarefas de programação, a sua interface deve ser de utilização fácil e intuitiva. A interacção do sistema com o utilizador é um factor importante a ter em conta na implementação deste sistema. Esta secção pretende tecer algumas considerações sobre as características das interfaces.

3.5.1 Características essenciais das interfaces de um SA

As características de uma interface para um sistema de animação de algoritmos foram largamente discutidas em [SDBP97]: para construir um sistema deste tipo, que pretende ser fácil de usar e rápido de aprender, o desenho da interface tem uma importância fundamental. Assim, a construção da interface deve seguir (segundo Stasko) os dez mandamentos da animação de algoritmos:

- Ser consistente. Deve haver consistência na representação e movimento dos objectos, ou seja, acções similares devem ser animadas de forma semelhante. Deve haver consistência na utilização do controlador VCR (janela mais activa que controla o andamento da animação ao longo do algoritmo).

- Ser interactivo. Um elevado grau de interactividade é necessário para manter o utilizador interessado e aumentar a sua capacidade de entendimento. Os utilizadores devem ser forçados a intervir cada 45 segundos da animação. Adicionalmente, o utilizador deve poder escolher o *input* e os iniciantes devem ter um *input* prédefinido. A navegação, ao longo da animação, passo a passo, é também de grande ajuda para o utilizador.

- Ser claro e consistente. Os pontos essenciais de cada algoritmo devem ser mostrados claramente e no instante em que se atravessa cada ponto. O sistema deve ter uma navegação *hierarquizada*: es-

conder detalhes mais complexos e permitir acedê-los carregando num botão do tipo `step into`. Esta técnica permite, em primeiro lugar, a aprendizagem de conceitos fundamentais e depois expandir para um maior grau de complexidade. O utilizador deve ter sempre a percepção de como a animação está relacionada com o algoritmo e ter sempre presente as posições do algoritmo entre as quais está a ser animado o seu funcionamento.

- Ser tolerante com o utilizador. O sistema deve ser generoso e tolerante para com os erros do utilizador. O sistema deve testar todas as acções, alertando o utilizador para situações de erro.

- Adaptar-se ao nível de conhecimento do utilizador. Deve permitir que o utilizador escolha a velocidade da animação. O sistema deve permitir parar a animação e recomeçá-la no mesmo ponto.

- Melhorar a componente visual da animação. Sendo a animação um processo gráfico, as visualizações não devem ter que ser elas próprias interpretadas. O facto de acções similares terem animações similares faz com que o utilizador reconheça os passos repetitivos mais facilmente.

- Manter o utilizador interessado. Uma animação não deve apenas captar o interesse inicial do utilizador mas deve mantê-lo atento e interessado durante toda a execução. A animação deve poder ser abortada a qualquer momento (é importante quando o algoritmo já foi compreendido mas continua a processar grandes quantidades de informação) e deve ser possível a colocação de pontos de paragem ao longo do algoritmo. Devem ser usadas anotações de voz e a animação deve usar bons exemplos do mundo real para explicar conceitos teóricos.

- Incluir representações simbólicas e icónicas. O utilizador deve poder ver em paralelo a execução do algoritmo como animação (icónica) e como pseudo-código (simbólica).

- Incluir análise de algoritmos. Sempre que apropriado, deve ser incluída a análise ao comportamento do algoritmo. Uma vez que o algoritmo pode *correr* várias vezes, seria útil para o estudante ser capaz de observar de que forma diferentes opções afectam a operação dos algoritmos. Estas opções poderão ser os próprios parâmetros do algoritmo e suas variações. As comparações visuais entre diferentes algoritmos similares proporcionam um entendimento mais profundo desses algoritmos.

- Incluir a história da execução. Uma animação mantém sempre uma espécie de historial que mostra porque é que a acção corrente está a acontecer, baseado-se no comportamento das acções passadas.

3.5.2 Interfaces para inserção de programas e escolha de visualizações

Para o sistema Alma seria desejável a construção das seguintes interfaces:

- editor para inserção do texto fonte
- janela de visualização da animação (representação visual da sequência de instruções e de conteúdos de variáveis)
- janelas para programação da animação incluindo uma janela de visualização da DAST para possível escolha dos nodos (partes do programa) a visualizar
- janelas para leitura das variáveis

Com excepção do editor, as restantes janelas são construídas pela *back-end* do Alma aquando da submissão de um programa fonte. Para o sistema Alma, numa primeira fase, é usado o editor de texto do sistema LISA para escrever o programa a animar. O ficheiro do programa fonte entra como argumento quando o ficheiro java do *back-end* é interpretado, dando origem a uma nova janela com a animação.
Numa segunda fase, prevê-se a construção de uma janela para introduzir o nome do programa a animar. Essa janela fará com que surjam outras duas janelas: uma com a árvore gerada pelo *front-end* e outra com alguns parâmetros de animação que o utilizador poderá controlar, nomeadamente: nível de detalhe da animação (frequência de amostragem) e partes do programa a visualizar. Durante esta programação da animação, poderão surgir janela para introdução de valores de variáveis (instruções de leitura).
Numa terceira fase,está previsto completar a janela de programação da animação com a facilidade de criação de novas regras de visualização ou de reescrita.
O sistema Alma permite diminuir a frequência de amostragem, diminuindo o número de passos da animação e consequentemente o seu nível de detalhe. O utilizador terá acesso a todas as visualizações da animação, pelo que poderá tirar partido da animação do programa quantas vezes desejar e à velocidade que pretender e em ambos os sentidos de execução das instruções. No entanto, não poderá interromper a animação para alterar parâmetros. Os parâmetros são definidos no início e depois todos os passos de animação são gerados de uma vez só. Mas poderá repetir a geração da animação com outros valores de parâmetros de animação ou com novos valores para as variáveis que são lidas ao longo do programa, com o objectivo de fazer análises comparativas de programas.
Os desenhos a usar na criação das animações devem ser muito simples, evitando que o desenho completo

de uma visualização se torne muito complexo. Será necessário definir desenhos que distingam as várias instruções e o respectivo grau de aninhamento.

Capítulo 4

Especificação do sistema Alma

Neste capítulo será apresentada a arquitectura do sistema Alma e serão explicadas as especificações de cada uma das partes dessa arquitectura. Para descrição das estruturas de dados e operações vai usar-se a notação típica de uma linguagem algoritmica imperativa (em português) tipo Pascal. As gramáticas serão escritas em BNF com o símbolo ':' a representar a operação de derivação (à moda do Yacc e outros). Serão ainda discutidas questões relacionadas com a representação intermédia do sistema, algoritmos de construção das animações e visualizações a gerar. Sobre esta parte do trabalho pode também ser consultado [VH00] e [VH01].

4.1 Arquitectura do sistema

O sistema é constituído pelo *front-end* de um compilador([WG84],[Cre98]) para reconhecer a estrutura do programa fonte (reconhecer símbolos e as regras de derivação; calcular atributos) e construir a árvore decorada. A arquitectura do sistema Alma está esquematizada na figura 4.1. Inspirados na tecnologia tradicional da compilação (bem conhecida e bem fundada), o princípio em que se vai basear o sistema Alma é de que a Árvore de Sintaxe Abstracta Decorada (DAST[1]) juntamente com a Tabela de Símbolos constituem o conjunto de estruturas de dados adequado para manter a informação relevante (nomes e endereços de variáveis e estruturas de controlo de fluxo) a um sistema que vai permitir a animação do algoritmo e a visualização do conteúdo de variáveis. Se pretendermos animar um algoritmo implementado noutra linguagem de programação, o *front-end* deverá ser alterado de forma a conseguir extrair dessa linguagem fonte a informação necessária para a animação. Mas tal alteração é sistemática, tendo por base a

[1] Do inglês, Decorated Abstract Syntax Tree

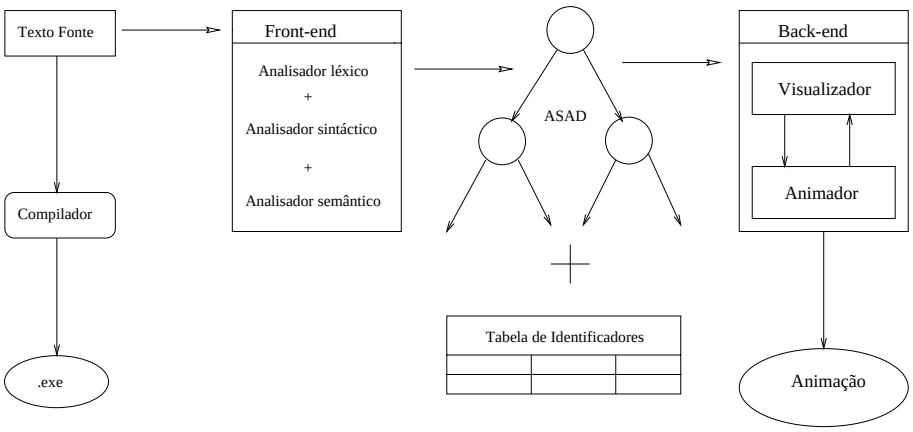

Figura 4.1: Arquitectura do Sistema Alma

gramática de atributos da linguagem fonte a analisar, e pode ser automatizada recorrendo às tradicionais ferramentas de geração de compiladores.

O sistema de animação constitui o *back-end* desse mesmo compilador e deve ser construído de forma genérica de modo a poder funcionar com qualquer *front-end*.

Em conclusão, a arquitectura proposta segue o método de implementação de compiladores designado por *tradução orientada à semântica*, em oposição à conhecida *tradução orientada à sintaxe*. A implementação é influenciada pelo uso de gramáticas de atributos para especificar a sintaxe e a semântica das linguagens. Seguindo este modelo, o significado do programa fonte é representado explicitamente numa árvore de sintaxe decorada com valores de atributos - DAST. Surge então uma separação clara entre *front-end* do compilador (responsável pela análise do programa e pela construção e decoração da árvore) e o *back-end* encarregado da tradução.

Para simular a execução, permitindo um controlo total sobre as variáveis e blocos a inspeccionar, a ideia chave é escolher o conjunto de atributos significativos para uma tarefa específica de visualização.

O *back-end* terá um conjunto de primitivas de visualização a associar a alguns dos nodos da árvore. A travessia da árvore (efectuada pelo *back-end*) tem por objectivo criar um desenho que represente (estaticamente) o programa a animar. Ao longo da travessia são usados atributos previamente calculados (para obter valores de variáveis e decidir a sequência de instruções a executar) e, é feito o agrupamento de figuras (locais aos nodos) de modo a obter-se o desenho completo do programa. Esse desenho, ao qual chamamos **global** é criado ao coleccionar as pequenas figuras associadas a determinados nodos, às quais

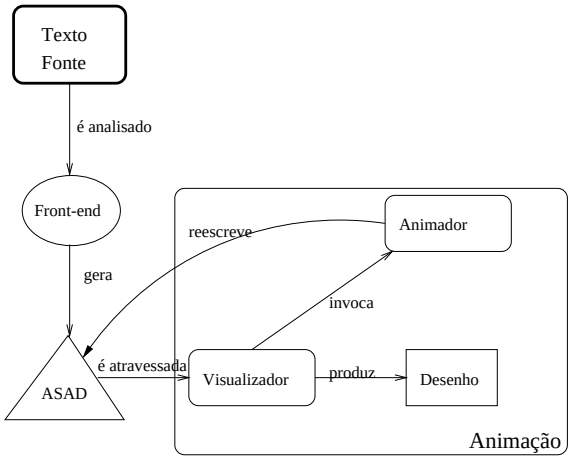

Figura 4.2: Linha de produção da animação a partir do texto fonte

chamamos **figuras locais**. Essa recolha é feita por uma travessia *Top-Down* (essencialmente *Post-fix*) da árvore, sintetizando-se o desenho global por agrupamento das figuras locais num processo *Bottom-Up*, tal como no trabalho descrito em [KA95].

Podemos então afirmar que o *back-end* será constituído por um *visualizador* e por um *animador*. O visualizador produz o desenho do programa num determinado instante e o animador aplica uma regra de reescrita de árvores à DAST (como no AGG [MRRT99]). Após cada reescrita, obtém-se um novo desenho à custa de uma outra travessia da árvore, agora com nova forma e com os valores dos atributos, eventualmente alterados, já actualizados. A apresentação dos sucessivos desenhos constitui a animação. As regras de reescrita permitem programar a animação não actuando directamente no desenho mas na árvore que produz esse desenho, sendo todo este processo independente do paradigma da linguagem fonte. A figura 4.2 mostra toda a linha de transformações entre o texto fonte e a animação pretendida.

Uma primeira travessia da árvore serve para criar uma representação visual do programa de entrada. Esta visualização nem sempre é pretendida porque pode não traduzir directamente a execução do programa mas apenas o programa em si. Em muitos casos, por exemplo, programas não sequenciais ou com recurso a subprogramas, a representação da execução (concordante com a sua semântica operacional) não coincide com a representação visual do programa textual completo. Um outro exemplo, é o caso dos programas em Prolog, que irá ser falado no capítulo 6. A animação de um processo de prova requer apenas a visualização dos predicados usadas nessa prova e não de toda a base de conhecimento.

É pois, necessário distinguir a árvore construída pelo *front-end* que corresponde ao texto fonte, da árvore

sobre a qual correrá a animação. Esta é obtida da primeira extraindo em cada nodo as subárvores correspondentes aos blocos de instruções que efectivamente não serão executados. A esta nova árvore chama-se *árvore de execução* ficando a árvore de entrada do sistema como sendo a*árvore de programa*. Assim sendo, o processo de construção da animação começa pela construção da*árvore de execução* e a primeira visualização traduzirá o primeiro estado de execução do programa (que corresponde à operação de *load* nos sistemas reais).

É de notar, que os algoritmos que implementam o visualizador e o animador são fixos (são sempre os mesmos para qualquer tipo de programa fonte), o que confere a generalidade pretendida ao *back-end* do Alma. Cada um desses algoritmos trabalha sobre informação contida em bases de regras separadas, facilmente alteráveis ou expansíveis, o que confere adaptabilidade ao sistema.

4.2 A DAST como representação intermédia

No sistema Alma é então usada uma DAST, como representação interna do significado do programa, isolando-se todas as dependências da linguagem fonte no *front-end*, mantendo o mecanismo de animação genérico no *back-end*. Isto permite concentrar as tarefas de reconhecimento e representação separadas da tarefa de animação. Esta abordagem torna o sistema Alma genérico.

A DAST é especificada por uma gramática abstracta independente da linguagem fonte. Essa gramática que define todos os tipos de nodos da DAST, será apresentada na secção 4.7. Assim, a DAST (enquanto *árvore de execução*) representa o estado do programa em cada momento e não reflecte directamente a sintaxe do programa fonte. Desta forma, a DAST será reescrita para descrever diferentes estados do programa simulando a sua execução. É de notar, que se trata de um processo de transformação semântica e não apenas uma reescrita sintáctica.

A estrutura de um nodo típico de uma árvore de *parsing* da gramática concreta está representada na figura 4.3. Cada nodo tem um identificador de produção (chave), um identificador do símbolo (à esquerda nessa produção), um conjunto de atributos associados ao símbolo do nodo e um conjunto de subárvores correspondentes aos símbolos do lado direito dessa mesma produção (descendentes, ou nodos filhos). O símbolo e o número da produção definem o tipo de nodo. Os nodos da DAST gerados pelo *front-end* serão conhecidos do *back-end*. Ou seja, o *back-end* tem uma lista de nodos que poderão ser gerados pelo *front-end* para os diversos tipos de linguagens. Acontece que a maioria dos tipos de nodos são comuns às várias linguagens. Como já foi dito, o *back-end* tem informação sobre a representação visual de cada tipo de nodo.

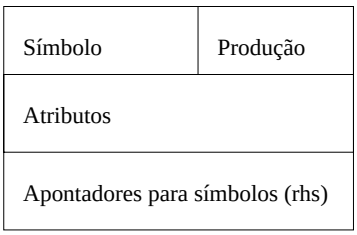

Figura 4.3: Estrutura dos nodos da DAST

4.2.1 Definição formal da árvore de sintaxe abstracta decorada

Formalmente, cada nodo da DAST tem a seguinte estrutura:

```
DAST = DASTNode
DASTNode = <  numProd: idProd,
              nomeSimb: idSimb,
              atributos: AlmaAtribs,
              filhos: seq(DASTNode)>
AlmaAtribs = <atnome: name,atvalor: value,
              attipo: type,attabId: tabId)>
```

A árvore vai ser representada por um nodo (raiz) que é identificado pelo número da respectiva produção da gramática e pelo axioma (ou símbolo inicial). A estrutura que representa esse nodo inclui também um tuplo (tipo, nome, valor, tabela_de_identificadores) que representa os atributos. Por último, essa estrutura inclui uma sequência de árvores que correspondem a todas as subárvores deste nodo. Os atributos associados a cada nodo são essenciais para sabermos qual o *objecto* representado e quais as suas características. O quarto atributo não está directamente relacionado com o *objecto* descrito, mas com a zona de código onde se encontra. Ou seja, pretende-se associar à zona do programa principal a tabela de identificadores global e à zona de um subprograma uma tabela local. Este atributo serve então para indicar qual a tabela a utilizar quando se reescreve este nodo.

Algumas funções foram criadas para aceder aos nodos da DAST e respectiva informação:

```
createNode: idProd X idSimb X AlmaAtribs
```

```
            X seq(DASTNode) -> DASTNode
setProdNumber: DASTNode X idProd ->DASTNode
getProdNumber: DASTNode -> idProd
getSymbol: DASTNode -> idSimb
setAttributes: DASTNode X AlmaAtribs -> DASTNode
getAttributes: DASTNode -> AlmaAtribs
setNodes: DASTNode X seq(DASTNode) -> DASTNode
getNodes: DASTNode -> seq(DASTNode)
size: DASTNode -> inteiro
equals: DASTNode X DASTNode -> Bool
```

Outras funções foram criadas para definir e aceder a cada um dos atributos:

```
setName : DASTNode X name -> DASTNode
setType : DASTNode X type -> DASTNode
setValue: DASTNode X value -> DASTNode
setTabId: DASTNode X tabId -> DASTNode
getName : DASTNode -> name
getType : DASTNode -> type
getValue: DASTNode -> value
getTabId: DASTNode -> tabId
```

4.2.2 Definição formal da Tabela de Identificadores

A Tabela de Identificadores conterá para cada identificador: o nome; o tipo (inteiro,float,...); a classe (VAR, CALLPROC, PROCDEF,...); o valor; o tipo de parâmetro que representa (de entrada, de saída ou ambos); e a localização na árvore.

```
tabId = ff(n: name,i: info)

info = <tp: type, classe: idSimb , val: value ,
        end: address, tpar:{OUT, IN, INOUT}>
```

As funções para uso desta tabela serão:

```
createTab: DASTNode -> tabId
putTableId: tabId X name X info -> tabId
putTableVal: tabId X name X value -> tabId
getTableType: tabId X name -> type
getTableClass: tabId X name -> idSimb
getTableVal: tabId X name -> value
getTableAddress: tabId X name -> address
getTableTpar: tabId X name -> {OUT, IN, INOUT}
```

4.3 Construção da DAST: o *front-end*

Existem dois tipos de utilizadores do sistema Alma : um que constrói o *front-end* para uma determinada linguagem (caso ainda não exista) e um outro que introduz um programa e recebe a sua animação automaticamente. Quem desenvolve o *front-end* tem apenas que conhecer a gramática da sua linguagem e efectuar uma extensão a essa gramática onde especifica o *mapping* conceptual entre as entidades da linguagens fonte e os nodos da gramática abstracta do Alma ou seja, indica as directivas de tradução entre a gramática concreta e a gramática abstracta.

Dado o conjunto de nodos possíveis na DAST representarem as operações mais básicas, comuns a muitas linguagens, torna-se fácil para o utilizador indicar quais desses conceitos estão envolvidos na sua linguagem e como se relacionam.

O *front-end*, gerado a partir da especificação dessas linguagens (gramáticas), vai construir a representação interna de um programa fonte. Essa representação é a DAST construída com base na GA do Alma. Os construtores dos nodos da DAST são especializações da função `createNode` que permitem construir nodos de vários tipos interligando-os em forma de árvore. A árvore é construída das folhas para a raiz. Os construtores dos nodos necessitam informação sobre o nodo a construir (símbolo, código do símbolo, número da produção e atributos) e seus descendentes. Os construtores das *folhas* da árvore apenas necessitam da informação sobre o próprio nodo. Considera-se que o nome do nodo é o nome do símbolo.

Os construtores são invocados nas acções semânticas associadas à gramática da linguagem fonte. A cada produção da gramática é associada a construção de um nodo da DAST que corresponde ao símbolo do lado esquerdo dessa mesma produção e que ficará como ascendente de todas as subárvores correspondentes aos símbolos do lado direito da produção.

```
gramática_do_FE = seq(produção)
produção = <lado_esq, lado_dir, semântica>
semântica = seq(construtor)
construtor = <nome, params>
```

Ao invocar um construtor está feito o mapeamento entre os conceitos definidos na gramática da nova linguagem e os conceitos pertencentes à gramática abstracta do Alma. Entendeu-se representar nesta gramática do Alma os conceitos mais básicos comuns às linguagens. Esses conceitos sendo básicos são mais gerais e servirão para definir conceitos mais alto nível das diferentes linguagens fonte. Em seguida serão apresentados dois exemplos de linguagens mapeadas nos nodos abstractos do sistema Alma.
Considerando uma primeira linguagem de estilo imperativo onde são permitidas apenas operações de atribuição onde o lado esquerdo da atribuição é sempre uma variável e o lado direito pode ser uma variável, um número inteiro, ou uma adição de inteiros. As primeiras produções da gramática serão especificadas da seguinte forma:

```
START : inicio STMTS fim  { ALMA_ROOT(START,STMTS)}
STMTS : STMT STMTS        { ALMA_STATS(STMTS_0,STMT,STMTS_1)}
      | STMT              { ALMA_STATS_SING(STMTS,STMT)}
```

Na primeira produção da gramática é usada uma função de construção da raiz da árvore. Esta linguagem permite definir várias instruções e o nodo do Alma usado para agrupar na árvore um conjunto de entidades do mesmo tipo, tem o nome `STATS`. Assim, será criado um nodo `STATS` com dois filhos, no caso de haver mais do que uma instrução, ou um nodo `STATS` com um filho, quando apenas há uma instrução. O nome das funções indica o tipo de nodo a criar (produção) e os parâmetros indicam o nome do símbolo do nodo que está a ser criado e os nodos filhos, cujas árvores já foram entretanto geradas. Uma outra produção da gramática será a seguinte:

```
STMT : ASSIGN  { ALMA_IDENT(STMT,ASSIGN)}
```

A função `ALMA_IDENT` não cria nodos novos apenas indica que a árvore associada ao nodo `ASSIGN` é a árvore a associar ao símbolo `STMT`. Uma atribuição será especificada pela seguinte produção:

```
ASSIGN : identificador = EXPR {ALMA_ASSIGN_VAR(ASSIGN,
                                getName(identificador),
                                getType(identificador),EXPR)}
```

A função aplicada a esta produção cria dois nodos: o nodo que representa a atribuição e o nodo que representa a variável atribuída. Esta função usa também a árvore já associada ao símbolo `EXPR`. O nome do identificador e o tipo são usados para criar o nodo que representa a variável atribuída. Uma expressão ficará definida pelas seguintes produções:

```
EXPR : EXPR + EXPR { ALMA_OPER(EXPR_0,EXPR_1,EXPR_2,''+'')}
EXPR : numero { ALMA_CONST(EXPR,getValue(numero), getType(numero))}
EXPR : identif { ALMA_VAR(EXPR,getName(identif),getType(identif))}
```

A produção que permite definir a adição de expressões levará à criação de um nodo `OPER` que simboliza, neste caso, a operação de adição e cujos filhos são as subárvores relativas aos operandos. Neste caso, uma expressão se não for uma adição é apenas um numero ou uma variável. Na primeira hipótese será criado um nodo `CONST` e, no segundo, um nodo `VAR`.

Considerando o programa exemplo abaixo e a gramática com as acções semânticas acima construída, pode-se deduzir que a DAST gerada pelo *front-end* é a representada na figura 4.4.

Exemplo 4.3.1 (Programa de entrada)

```
inicio
a = 5
b = a + 3
fim
```

A árvore gerada será a da figura 4.4.

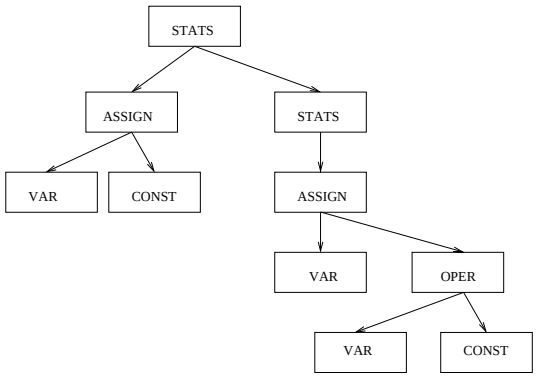

Figura 4.4: A DAST gerada para o exemplo 4.3.1

Um segundo exemplo de mapeamento mostra o caso de outro tipo de linguagem para a qual também é possível identificar conceitos que podem ser representados pelos nodos da DAST do Alma. A linguagem pretende descrever o funcionamento de um conjunto de máquinas de transição de estados, indicando, para cada máquina: o nome, o estado inicial e um conjunto de transições de estados sujeitas a pré-requisitos (condições). A gramática começa pelas seguintes produções:

```
Maquinas : Maquina Maquinas
           { ALMA_STATS (Maquinas,Maquina,Maquinas)}
         | Maquina
           { ALMA_STATS_SING (Maquinas,Maquina)}
```

Pretende-se nestas primeiras produções definir um conjunto de *entidades* do mesmo tipo: sendo assim, o nodo a criar é o STATS, quer para o caso de ser mais do que uma máquina, quer no caso de ser apenas uma. A informação relativa a cada máquina é especificada da seguinte forma:

```
Maquina    : Nome Estado_ini Transições
             {ALMA_LST(Nome, Estado_ini, Transições)}
Transições: Transição Transições
             {ALMA_STATS(Transições_0,Transição,Transições_1)}
           | Transição
             {ALMA_STATS_SING(Transições, Transição)}
```

Ao contrário do nodo STATS, o nodo LST representa um associação de entidades de tipos diferentes. Assim, todos os conceitos relacionados com uma máquina ficam agrupados por LST. A cada máquina está associada também um conjunto de transições, sendo cada transição representada pela produção:

```
Transição  : EstadoA  Condição  ->  EstadoB
             {ALMA_IF(Transição, Condição,
               ALMA_STATS(ALMA_OPER(EstadoA,EstadoB,''transita'')))}
```

Cada transição poderá ser mapeada num nodo IF pois também está sujeita a uma condição. Se a condição é verificada a operação é executada, ou seja, a máquina "transita" de um estado A para um estado B. As últimas produções da gramática são especificadas da seguinte forma:

```
Nome,
EstadoA,
EstadoB    : identif
             {ALMA_CONST(...,getName(identif),getType(identif))}
Condição   : Var_teste = Val_teste
             {ALMA_OPERREL(Condição,Var_teste,Val_teste,''=='')}
Var_teste  : identif
             { ALMA_VAR(Var_teste,getName(identif),getType(identif))}
Val_test   : numero
             { ALMA_CONST(Val_teste,getValue(numero),getType(numero))}
```

Cada estado é visto como um nodo `CONST` pois a única informação que tem associada é um nome e o mesmo acontece com o nome da máquina. Cada condição tem o formato de uma comparação de igualdade entre um identificador e um número inteiro. As condições serão mapeadas para nodos do tipo `RELOPER` cujos filhos serão do tipo `VAR` e `CONST` respectivamente.

Tomando como exemplo o seguinte texto de entrada, escrito na linguagem especificada pela gramática:

Exemplo 4.3.2 (Programa de entrada)

```
ROLDANA
A
A a=3  -> B
B a=1  -> A
```

O *front-end* construído para esta linguagem produzirá, como resultado, a DAST da figura 4.5.

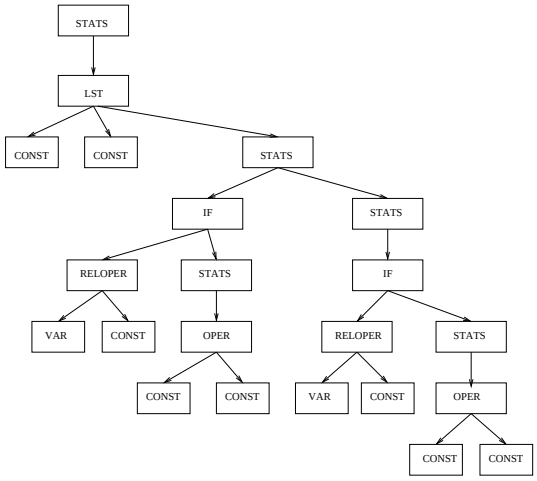

Figura 4.5: A DAST gerada para o exemplo 4.3.2

4.4 Construção da animação: o *back-end*

A função que despoleta todo o processo de construção da animação, `animate()`, sintoniza o processo de reescrita com o processo de construção de uma visualização. O processo de reescrita simula a execução do programa e deve ser interrompido sempre que uma visualização deve ser recolhida. Se a DAST for desenhada cada vez que é alterada teremos como resultado um maior número de visualizações e, portanto, uma animação mais detalhada. Dependendo do tipo de visão que se pretende obter do

programa a função `visualize()` será invocada mais ou menos vezes. Assim, essa frequência de amostragem poderá ser controlada pela função `shownow()` como se mostra a seguir:

```
animate(tree: DASTNode){
        visualize(tree);
        Do    rewrite(tree);
              If shownow()
              then visualize(tree);
        until (tree==rewrite(tree))
}
```

Quando nenhuma regra de reescrita se aplica à árvore, esta não é modificada e termina o processo de reescrita e consequentemente o processo de visualização.
O momento de invocação da função `visualize()` depende da escolha do utilizador e de um conjunto de informações sobre tipos de nodos e historial de modificações da árvore. O sistema deve usar essas informações para decidir como sintonizar os processos de reescrita e visualização. Por exemplo, poderão existir alguns nodos cuja modificação não é pretendida num determinado tipo de visualização, assim a função `visualize()` não seria invocada enquanto se estivesse a efectuar a reescrita desses nodos.
O processo de visualização e o processo de reescrita serão descritos com mais pormenor nas próximas secções.

4.5 Visualização no sistema Alma

A visualização é conseguida aplicando regras de visualização às subárvores da DAST; a especificação dessas regras define o *mapping* entre árvores e figuras. Juntando todas essas figuras parciais é criada a representação visual do programa.

4.5.1 Regras de visualização

A base de regras de visualização (VRB [2]) estabelece a correspondência (*mapping*) entre cada árvore e um conjunto de pares. Cada par (uma regra de visualização) é composto por uma condição (`cond`) e uma função que define a representação visual da árvore (`dp`).

$$\text{VRB: DASTNode} \mapsto \text{set (condition} \times \text{drawing\_procedures)}$$

[2] Do inglês, Visualizing Rule Base

Cada condição é um predicado sobre valores de atributos associados aos nodos da respectiva árvore, que restringe o uso do procedimento de desenho (a aplicabilidade da regra). Na prática, cada regra de visualização contem ainda um terceiro elemento que representa a árvore a visualizar, sendo especificada da seguinte forma:

```
vis_rule(idProd)= <t: tree-pattern>,
                  (cond: condition),
                  {dp: drawing_procedures}

tree-pattern = <root, child_1, ..., child_n>
```

`cond` é uma expressão booleana (por defeito, `true`) e o procedimento de desenho `dp` é uma sequência de uma ou mais chamadas a procedimentos elementares de desenho.
Uma regra de visualização poderá ser aplicada a todas as árvores que são instâncias da produção `idProd`. Uma árvore padrão é especificada através da `tree-pattern`, usando variáveis que representam cada nodo. Cada nodo tem, entre outros, atributos do tipo `type`, `name` e `value` que serão usados na especificação das regras, quer para formular condições, quer para serem passadas como parâmetros para o procedimento de desenho. Poderá haver mais do que uma regra para a mesma produção da gramática, para a mesma subárvore.
Por exemplo, considerando as produções:

```
popr:    reloper      : exp exp
pexp1:   exp          : CONST
pexp2:   exp          : VAR
pexp3:   exp          : oper
pexp4:   exp          : reloper
```

para construir a representação visual deste operador relacional, considera-se, a título de exemplo, que são necessárias duas regras: uma para o caso do valor das variáveis não ser conhecido e outro para o caso contrário. Cada regra gera desenhos distintos como pode ser visto na figura 4.6, onde se considerou um operando variável e outro constante.
As regras a aplicar para gerar esses desenhos serão, respectivamente:

```
vis_rule(popr) =
    <opr: reloper,a: exp,c: exp>,
```

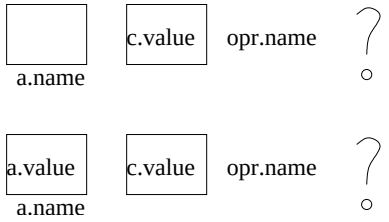

Figura 4.6: Visualização de uma operação relacional

```
     ((getValue(a)=NULL) AND
      (getType(a)=VAR) AND (getType(c)=CONST)),
     {drawRect(getName(a)),drawRect(),put(getName(opr)),
      put('?')}
vis_rule(popr) =
     <opr: reloper,a: exp,c: exp>,
     ((getValue(a)!=NULL) AND
      (getType(a)=VAR) AND (getType(c)=CONST)),
     {drawRect(getName(a),getValue(a)),drawRect(getValue(c)),
      put(getName(opr)),put('?')}
```

4.5.2 Algoritmo de construção das visualizações

O algoritmo de construção da visualização consiste em percorrer a árvore nodo a nodo tentando aplicar alguma regra de visualização. No entanto, devem ser aplicadas em primeiro lugar as regras que mapeiam os nodos filhos e depois num processo bottom-up (post-fix) aplicar as restantes. Para cada nodo é extraído da base de regras (VRB) o conjunto de regras de visualização que lhe está associado, verificando o identificador da produção. Depois verifica-se as restrições usando os valores dos atributos do nodo em causa. Por último, executam-se as instruções de desenho que constam do lado direito da regra de visualização seleccionada.

```
visualize(tree: DASTNode){
   If not(empty(tree))
   then forall t in children(tree)
           do visualize(t);
        rules <- VRB[getProdNumber(tree)];
        found <- false;
```

```
        while not(empty(rules)) and not(found)
           do r <- choice(rules);
              rules <- rules - r;
              found <- match(tree,r)
        If (found) then draw(tree,r);
}
```

4.6 Animação no sistema Alma

Cada regra de reescrita especifica uma transição de estados no processo de execução do programa; o resultado de aplicar uma regra é uma nova DAST obtida por uma mudança semântica ou sintáctica. Esta reescrita sistemática da DAST original é intercalada com uma sequência de visualizações produzindo assim a animação. A função principal sincroniza o processo de reescrita com o processo de visualização, de uma forma parametrizada e permitindo a geração de diferentes vistas para o mesmo programa fonte, conforme o algoritmo apresentado na secção 4.4 atrás.

4.6.1 Regras de reescrita

A base de regras de reescrita (RRB [3]) faz o *mapping* entre cada árvore e um conjunto de tuplos:

$$\text{RRB: DASTNode} \mapsto \text{set(condition} \times \text{tree-pattern} \times \text{attributes\_evaluation)}$$

Cada tuplo contem uma condição `cond`, uma nova árvore representada por `newtree` que define as transformações sintácticas e um procedimento de cálculo de atributos `eval`, que define as modificações semânticas. Na prática acrescenta-se um primeiro elemento que representa o nodo inicial a reescrever. A forma de escrever estas regras obedecerá à seguinte sintaxe:

```
rule(idProd)= <t:tree-pattern>,
              (cond:condition),
              <newProdId: idProd: newtree: tree-pattern>,
              {eval: attributes_evaluation}


tree-pattern = <root, child_1, ..., child_n>
```

Nesta especificação, `cond` é uma expressão booleana (por defeito, true) e o cálculo de atributos `eval` é um conjunto de instruções que definem os novos valores dos atributos (por defeito, skip). Uma regra

[3] Do inglês, Rewriting Rule Base

de visualização poderá ser aplicada a todas as árvores que são instâncias da produção `idProd`. A nova árvore é também uma produção da gramática abstracta, por isso, será especificado o identificador da nova produção. Uma árvore padrão `t` associa variáveis aos nodos para que possam ser usadas nos outros componentes da regra. Quando uma variável surge tanto na árvore padrão como na nova significa que toda a informação contida nesse nodo, incluindo atributos, será mantida.
Poderá haver mais do que uma regra para a mesma produção da gramática (para a mesma subárvore), sendo necessário seleccionar a regra aplicável (tal como se fez na VRB na secção 4.5).
Considerando, a título de exemplo, as seguintes produções pertencentes à gramática abstracta do Alma para definir uma instrução condicional:

```
pifelse:     IF     : cond actions actions
pifthen:            | cond actions
```

A DAST será modificada usando as seguintes regras:

```
rule(pifelse) = <se: IF,op: cond,a: actions,b: actions>,
                (getValue(op)=true),
                <pifthen:se: IF,op: cond,a: actions>,
                { }

rule(pifelse) = <se: IF,op: cond,a: actions,b: actions>,
                (getValue(op)=false),
                <pifthen:se: IF,op: cond,b: actions>,
                { }
```

A representação gráfica destas duas regras de reescrita é mostrada na figura 4.7. Os nodos `THEN` e `ELSE` representam os blocos `actions`.

4.6.2 Algoritmo de reescrita

O algoritmo de reescrita efectua travessias na árvore tentando aplicar alguma das regras de reescrita e terminará todo o processo se mais nenhuma regra puder ser aplicada. Para cada nodo, o algoritmo determina o conjunto possível de regras usando o identificador da produção e verificando as condições de contexto associadas a essas regras. A DAST será modificada substituindo o nodo mapeado em causa pela árvore especificada no lado direito da regra. Esta transformação pode ser só semântica (só mudam os valores dos atributos) mas também pode ser sintáctica. As alterações sintácticas implicam (normalmente) remoção de nodos, ou seja, terão que desaparecer todos os nodos descendentes do ramo removido. Este

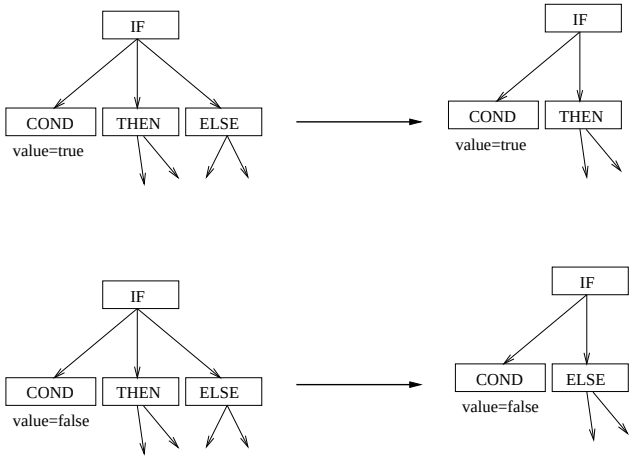

Figura 4.7: Regras de reescrita para instrução condicional

algoritmo segue uma abordagem *Top-Down* (ao contrário do algoritmo de visualização), ou seja, tenta aplicar regras ao nodo pai e só depois é que tenta reescrever os filhos (da esquerda para a direita). Sempre que ocorre uma reescrita, a árvore é redesenhada.

O algoritmo de reescrita mais detalhado será:

```
DAST rewrite(tree: DASTNode){
  If not(empty(tree)) then
     rules <- RRB[getProdNumber(tree)];
     found <- false;
     while not(empty(rules)) and not(found)
     do r <- choice(rules);
        rules <- rules - r;
        found <- match(tree,r)
     If (found)
     then tree <- change(tree,r)
     else a <- nextchild(tree)
          while not(empty(a)) and not(rewritten(a))
          do a <- nextchild(tree)
          If not(empty(a))
          then tree <- rebuild(tree,a,rewrite(a))
     return(tree)
}
```

```
bool rewritten(t){ return(t != rewrite(t))}

DAST rebuild(t,a,b){ t.a <- b; return( t ) }

DAST change(tree,<<maptree>,<cond>,
                <newtree>,<eval>>){
     instantiate(maptree,tree)
     build(newtree,maptree)
     evaluate(newtree,eval)
     return(newtree)
}
```

4.7 Gramática Abstracta e Bases de Regras

Após descrever a finalidade de cada módulo e o seu princípio de funcionamento (algoritmo), vai-se especificar, nesta secção, o tipo de nodos que uma DAST pode conter e as bases de regras sobre as quais trabalham os algoritmos de visualização e de reescrita.

4.7.1 Gramática da representação interna do Alma

A gramática abstracta, subjacente à construção da representação intermédia, contem os símbolos que identificarão os nodos da DAST. No entanto, para simplificar essa representação, nem todos os símbolos têm representação na árvore. Apenas os escritos em letras maiúsculas correspondem a nodos da DAST, os restantes são usados apenas para agrupar alternativas ou facilitar a leitura da gramática (símbolos escritos em letras minúsculas).

```
pprog:     prog            : STATS
pstats:    STATS           : stat STATS
pstat:                     | stat
pstat1:    stat            : IF
pstat2:                    | WHILE
pstat3:                    | ASSIGN
pstat4:                    | READ
pstat5:                    | WRITE
```

```
pstat6:                  | CALLPROC
pstat7:                  | PROCDEF
pstat8:                  | RETURN
pifelse:   IF            : cond actions actions
pifthen:                 | cond actions
pwhile:    WHILE         : cond actions
passign:   ASSIGN        : VAR exp
pread:     READ          : VAR
pwrite1:   WRITE         : VAR
pwrite2:                 | CONST
pcall1:    CALLPROC      : LST
pcall2:                  | LST PROCDEF
pcond:     cond          : RELOPER
pactions:  actions       : STATS
popr:      RELOPER       : exp LST
pop:       OPER          : exp LST
plst:      LST           : exp LST
pnull:                   | epsilon
pexp1:     exp           : CONST
pexp2:                   | VAR
pexp3:                   | OPER
pexp4:                   | RELOPER
pexp5:                   | CALLPROC
pconst:    CONST         : num
pret:      RETURN        : exp
ppdef:     PROCDEF       : LST STATS
pvar1:     VAR           : id
```

Cada programa será composto por grupos de instruções, predicados, funções e cada uma destas entidades é representada pelo símbolo `stat`. Cada `stat` pode representar vários conceitos: estruturas condicionais (`IF`) ou repetitivas (`WHILE`), atribuições (`ASSIGN`), leituras (`READ`) e escritas (`WRITE`),

invocação de subprogramas (`CALLPROC`), definição de subprogramas (`PROCDEF`) ou instruções de retorno (`RETURN`). A produção `pcall2` terá, neste momento, uma leitura menos natural (óbvia) porque resulta da aplicação de uma regra de reescrita, conforme se explica à frente (secção 4.8).
São consideradas, nesta primeira especificação, as variáveis do tipo simples, onde cada variável é representada por um identificador. No entanto, deverão ser acrescentadas mais três produções à gramática, de modo a abranger os tipos apontadores e estruturados.

```
pvar1:    VAR            : id
pvar2:                   | id
pvar3:                   | id id
pvar4:                   | id exp
```

A produção `pvar1` representa então as variáveis do tipo simples. A produção seguinte representa uma variável do tipo apontador também representada por um identificador. A terceira produção representa tipos estruturados onde o segundo identificador representa um campo da estrutura. A quarta e última produção, representa as variáveis indexadas onde poderemos ter como índice uma constante, uma variável ou uma expressão (desde que retorne um valor numérico).

4.7.2 Regras de reescrita

Nesta secção, serão apresentadas as regras de reescrita do sistema (que fazem parte da RRB), à excepção daquelas que se relacionam com os subprogramas porque serão explicadas à parte na secção 4.8. Algumas regras de reescrita serão apresentadas graficamente nas figuras 4.8 e 4.9. As regras reescrevem a árvore efectuando cálculo de atributos, alterando os atributos dos nodos e removendo/substituindo nodos. No entanto, será necessário utilizar a Tabela de Identificadores guardar os valores das variáveis definidas ou lidas para que todas as instruções onde são usadas possam aceder aos seus valores actualizados.

Como se disse atrás, na especificação de cada árvore (subárvore) irão ser usadas variáveis que representam os vários nodos. Quando a variável se mantém na árvore do lado direito significa que todos os atributos, apontadores e identificadores desse nodo se mantêm. Os cálculos especificados indicam as alterações dos atributos de uns nodos para outros. A árvore resultante da aplicação da regra está obrigatoriamente representada na gramática abstracta da linguagem fonte e, por isso, é indicado na própria regra qual a produção da nova árvore.

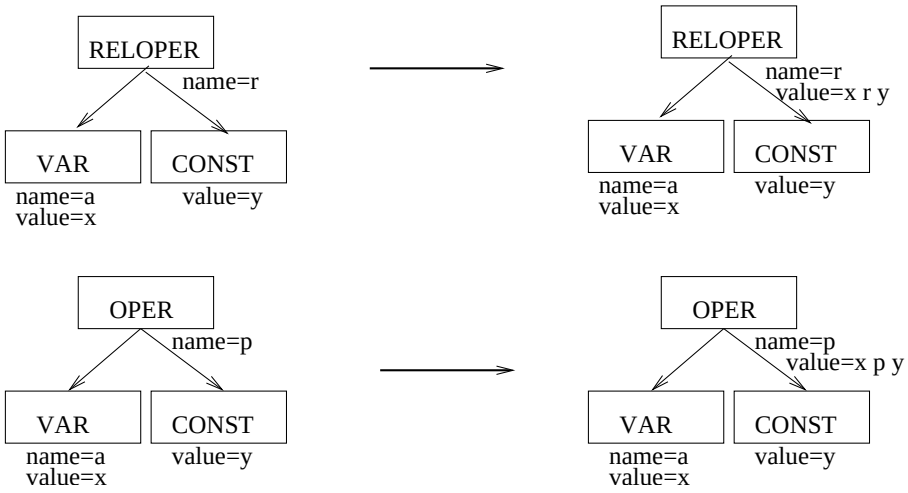

Figura 4.8: Regras de reescrita (Cálculo de operações)

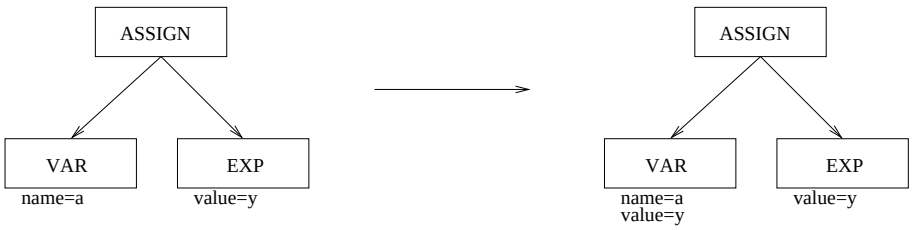

Figura 4.9: Regras de reescrita (Cálculo de atribuições)

É de notar que, uma variável da regra pode ser de um tipo cujo símbolo não tem representação na DAST, ou seja, trata-se de um símbolo pertencente à gramática abstracta do Alma, usado para agrupar alternativas mas não será criado nenhum nodo desse tipo. Na DAST apenas aparecerá o seu nodo descendente porque é neste que se encontram os atributos que interessa usar e calcular. No entanto, na especificação da regra faz sentido usar estes símbolos porque permite ter apenas uma regra para todos as alternativas que esse símbolo representa.

```
rew_rule(popr) = <op1: reloper,a: exp,b: exp>,
                 ( ),
                 <popr:op2: reloper,a: exp,b: exp>,
                 {setValue(op2,exec(getName(op1),getValue(a),getValue(b))}

rew_rule(pop) = <op1: oper,a: exp,b: exp>,
                ( ),
                <pop:op2: oper,a: exp,b: exp>,
                {setValue(op2,exec(getName(op1),getValue(a),getValue(b))}

rew_rule(passign) = <at: assign,a1: var,b: exp>,
                    ( ),
                    <passign: at: assign,a2: var,b: exp>,
                    {setName(a2,getName(a1));
                    setValue(a2,getValue(b))}

rew_rule(pifelse) = <se: if,op: cond,a: actions,b: actions>,
                    (getValue(op)=true),
                    <pifthen:se: if,op: cond,a: actions>,
                    { }

rew_rule(pifelse) = <se: if,op: cond,a: actions,b: actions>,
                    (getValue(op)=false),
                    <pifthen:se: if,op: cond,b: actions>,
                    { }

rew_rule(pifthen) = <se: if,op: cond,a: actions>,
                    (getValue(op)=false),
                    <>,
```

```
                    { }

rew_rule(pwhile) = <inst: while,cond1: cond,accao1: stats>,
                   (getValue(cond1)=true),
                   <pwhile: inst: while,cond2: cond,accao2: stats>,
                   {setName(accao2,``REP'')}

rew_rule(pwhile) = <inst: while,cond1: cond,accao1: stats>,
                   (getValue(cond1)=false),
                   <pwhile:inst: while,cond2: cond,accao2: stats>,
                   {setName(accao2,``SKIP'')}

rew_rule(pread) = <r: read,a: var>,
                  (getValue(a)=null),
                  <pread: r: read,a: var>,
                  {setValue(a,read())}

rew_rule(pwrite) = <w: write,a: var>,
                   (getValue(a)!=null),
                   <pwrite: w: write,a: var>,
                   {write(getValue(a))}
```

4.7.3 Regras de visualização

Nesta secção, serão apresentadas as regras de visualização pertencentes à VRB, à excepção das relacionadas com os subprogramas que serão explicadas na secção 4.8. Algumas regras de visualização serão apresentadas graficamente nas figuras 4.10 e 4.11. Para cada grupo de nodos são especificadas instruções de desenho que permitem criar a representação visual desses nodos. Neste caso, também é possível usar tipos de nodos sem representação directa na DAST mas que permitem referir de uma só vez todas as alternativas às quais pode ser aplicada a regra em causa. No entanto, embora cada regra agrupe vários casos possíveis, as figuras 4.10 e 4.11 representam casos mais específicos.

As funções de desenho, associadas a cada regra, foram definidas através das seguintes primitivas de desenho:

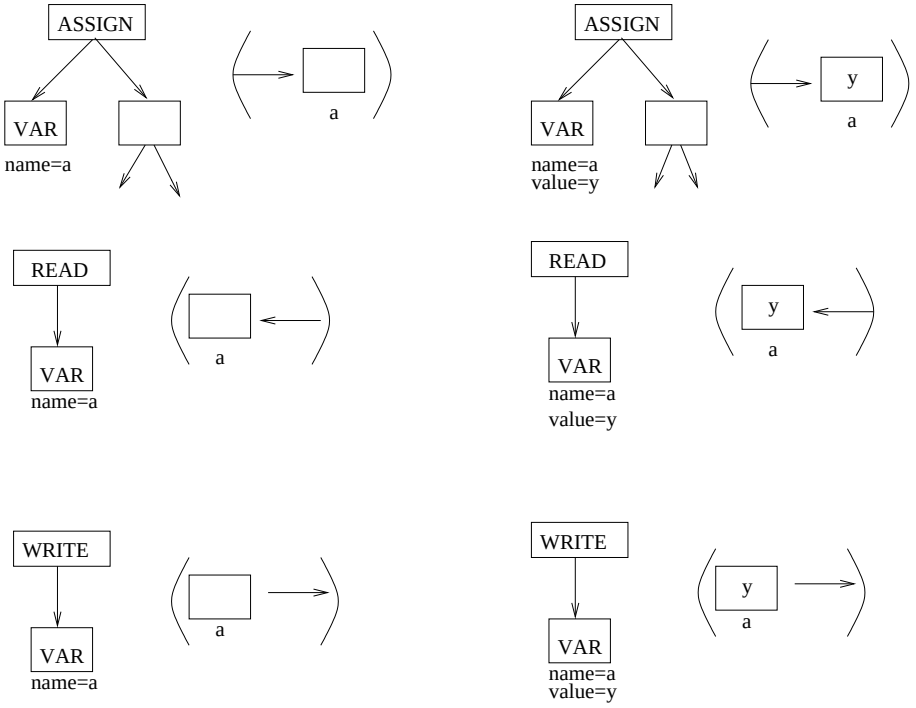

Figura 4.10: Regras de visualização (atribuição e entrada/saída)

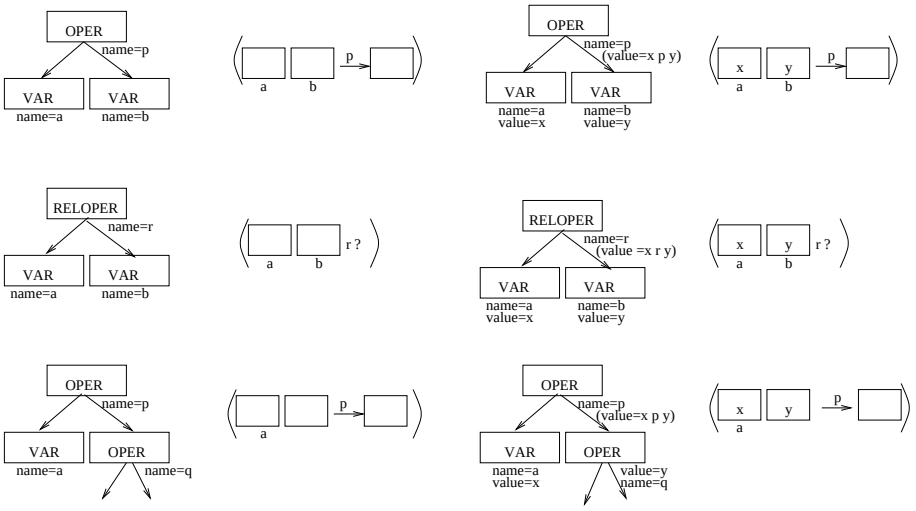

Figura 4.11: Regras de visualização (operações)

```
drawRect(OPT a:label; OPT v:value)
left_arrow()
put(a: char)
right_arrow(OPT a:label)
```

NOTA: Recorde que todos os nodos do tipo VAR, OPER e RELOPER têm associado um atributo nome.

```
vis_rule(passign)= <atrib: assign,a: var,b: exp>,
                    (),
                    {right_arrow(),drawRect(getName(a),getValue(b))}

vis_rule(pread)= <r: read,a: var>,
                 (),
                 {drawRect(getName(a),getValue(a)),left_arrow()}

vis_rule(pwrite)= <w: write,a: var>,
                   (),
                   {drawRect(getName(a),getValue(a)),right_arrow()}

vis_rule(pop) = <op: oper,a: exp,b: exp>,
                (((getType(a)=VAR)||(getType(a)=CONST)||
                (getType(a)=OPER)||(getType(a)=RELOPER)) AND
                ((getType(b)=VAR)||(getType(b)=CONST)||
                (getType(b)=OPER)||(getType(b)=RELOPER))),
                {drawRect(getName(a),getValue(a)),
                 drawRect(getName(b),getValue(b)),
                 right_arrow(getName(op))}

vis_rule(plst) = <lista: lst,a: exp,lista: lst>,
                  ((getType(a)=CONST)||(getType(a)=VAR)||
                  (getType(a)=OPER)||(getType(a)=RELOPER)),
                  {drawRect(getName(a),getValue(a))}

vis_rule(popr) = <opr: reloper,a: exp,b: exp>,
                  (((getType(a)=VAR)||(getType(a)=CONST)||
```

```
(getType(a)=OPER)||(getType(a)=RELOPER)) AND
((getType(b)=VAR)||(getType(a)=CONST)||
(getType(b)=OPER)||(getType(a)=RELOPER))),
{drawRect(getName(a),getValue(a)),
 drawRect(getName(b),getValue(a)),
 put(getName(opr)),put('?')}
```

4.8 Animação de subprogramas

A animação de programas deve contemplar também a visualização das instruções referentes a subprogramas. Tomando como exemplo o seguinte excerto de código em C:

Exemplo 4.8.1 (Programa com subprogramas)

```
int soma(int x,int y){
    return(x+y);
}
main(){
int z,a,b;
    a=2;
    b=3;
    z=soma(a,b);
    printf(``%d'',z);
}
```

A DAST gerada pelo sistema Alma para exemplo 4.8.1 é mostrada na figura 4.12.

A definição do subprograma pode surgir antes ou depois do programa que o invocou. Quer a invocação quer a definição do subprograma terão uma representação na DAST. O nodo `CALLPROC` representa a invocação de subprogramas, o nodo `LST` irá representar a lista de parâmetros de invocação e um novo nodo `PROCDEF` encabeça todos os nodos dos parâmetros e das instruções do subprograma.

Depois de criada a DAST surgem algumas dificuldades na definição das regras de reescrita que permitam a execução de um pedaço de código que apenas é *activado* quando ocorre uma invocação. Esse código pode ser usado zero, uma ou mais vezes durante o programa principal mas a sua definição surge normalmente num ponto do programa que nada tem a ver com a sua invocação. Neste caso, não interessa obter

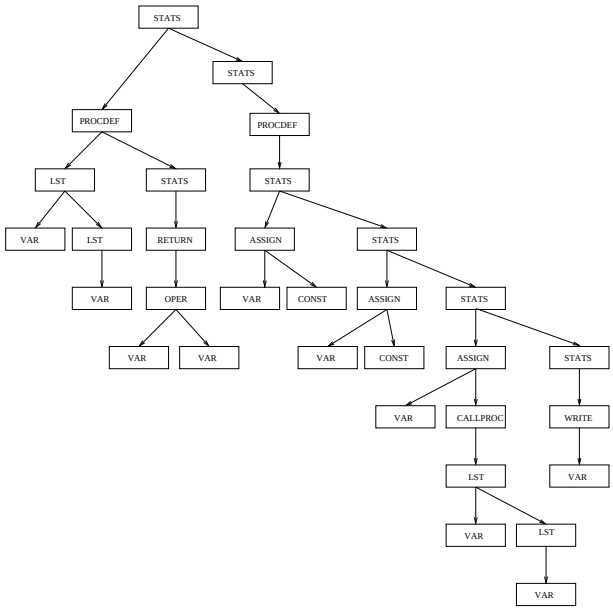

Figura 4.12: Representação abstracta de subprogramas

a visualização de toda a DAST mas apenas de parte (programa principal). No entanto, quando surge uma invocação é necessário visualizar a execução desse subprograma antes da visualização do restante programa. Embora seguindo a mesma abordagem será necessário solucionar este problema tendo em conta que o que se pretende visualizar os estados de execução de um programa e não o programa em si. Pensou-se numa primeira solução onde um nodo de invocação despoletava o processo de reescrita e visualização do subprograma *saltando* para essa subárvore da DAST e activando esses nodos até então adormecidos. Esta solução, no entanto, mostrou-se muito deficiente na medida em que a visualização do subprograma apareceria algures no meio da visualização do programa principal e, para além disso, não soluciona o problema da recursividade permitindo criar apenas uma instância de execução que trabalha sobre os nodos de definição do subprograma.

Assim sendo, e pegando novamente no conceito de árvore de execução, concluímos que a sua construção será responsável por colocar uma cópia dos nodos relativos ao subprograma como descendentes do nodo de invocação. A ideia, que já foi apresentada nas secções anteriores, é criar uma árvore de execução a partir da árvore de programa, que será composta apenas pelos nodos que efectivamente intervêm na execução do programa (de acordo com a semântica operacional). Esta árvore de execução conterá tantas cópias da subárvore de definição do subprograma quantas as invocações. Uma função recursiva será representada por várias instâncias do mesmo subprograma permitindo guardar os valores das variáveis

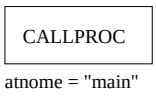

Figura 4.13: A primeira árvore de execução para o exemplo 4.8.1

(estado do programa) de uma invocação para outra.

A árvore de programa representa o programa de entrada contendo todas as definições de subprogramas (que poderão não ser usadas) e o programa principal. Trata-se de uma representação que será usada para criar a árvore de execução. A construção da árvore de execução implica a consulta da Tabela de Identificadores para determinar qual o subprograma invocado e tomar conhecimento dos nodos que deverão ser copiados. Ao considerarmos a própria função principal como um subprograma, a árvore de execução, numa primeira fase, é apenas constituída pelo nodo de invocação da função `main`. A figura 4.13 mostra, então, a primeira árvore de execução que é sempre a mesma, independentemente do programa a animar. Após algumas reescritas, as definições das funções são copiadas para a árvore de execução, ficando como é apresentado na figura 4.14.

Copiando os nodos de definição da função `main`, o nodo `PROCDEF` (também ele uma cópia) terá a referência da tabela de identificadores da função que, neste caso, poderá ser considerada a tabela de identificadores global do programa. O nodo `CALLPROC` não é removido da árvore porque permite guardar referência dos parâmetros de invocação e conterá o valor do subprograma depois da sua execução.

É de notar que em qualquer nodo da árvore há um atributo que contem uma referência para a tabela de identificadores a utilizar. Nos nodos relativos ao programa principal estará a referência para a tabela de identificadores global e nos nodos relativos aos subprogramas estará a referência da respectiva tabela local.

Reescrevendo os nodos relativos à função `main`, um outro nodo `PROCDEF` e respectivos descendentes são copiados para a árvore de execução, aquando da invocação do subprograma. No momento da invocação é consultada a tabela de identificadores para localizar a definição do respectivo subprograma; é criada uma tabela local com os valores dos parâmetros e variáveis locais (constam no nodo de declarações do subprograma); a tabela cujo endereço ficará guardado no nodo `PROCDEF`, é então actualizada com os valores dos parâmetros de invocação; é feita uma cópia da subárvore de definição que irá ficar *pendurada* no nodo `CALLPROC`. Este nodo contem então o endereço da tabela de identificadores externa ao subprograma e o nodo `PROCDEF` que será seu descendente terá associado o endereço da tabela local, o

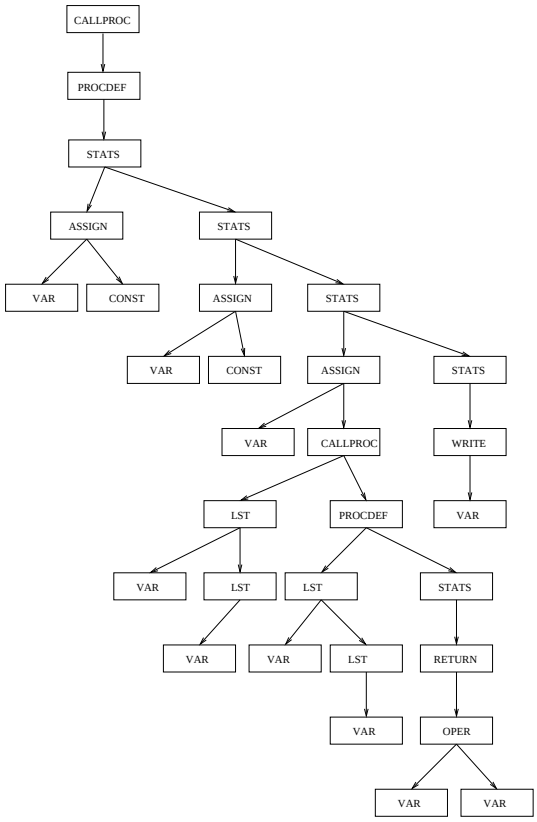

Figura 4.14: Árvore de execução para o exemplo 4.8.1

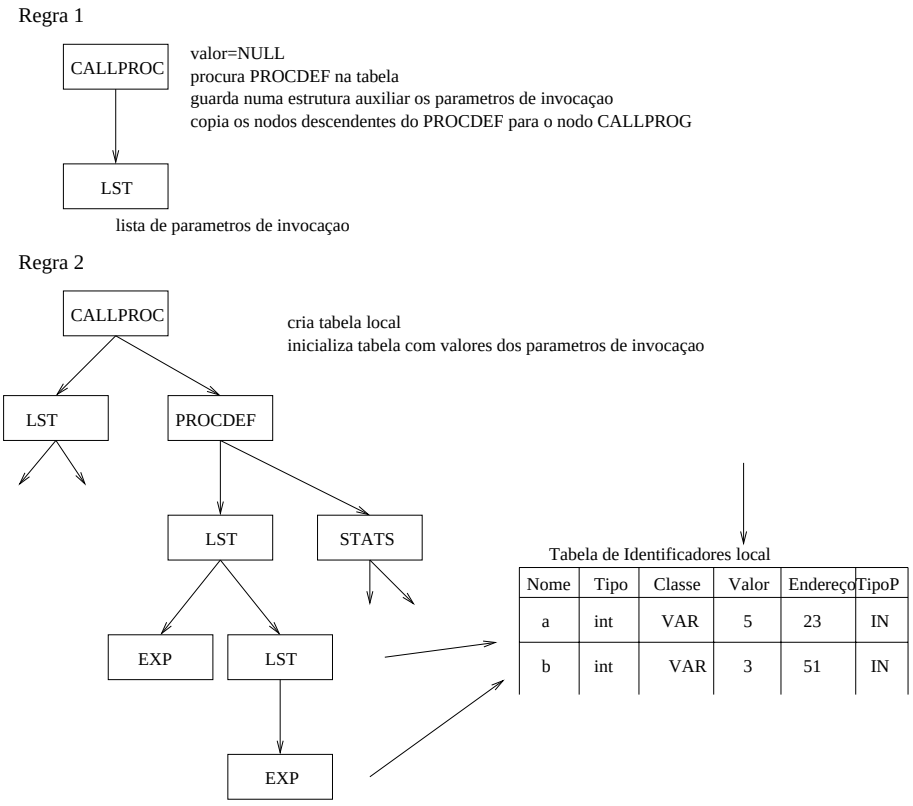

Figura 4.15: Regras de reescrita para invocação de subprogramas

que permitirá a actualização da tabela mais externa quando a local desaparecer.

A árvore de execução completa, apresentada na figura 4.14, contem todos os nodos do programa e do subprograma e as próximas reescritas irão reescrever os nodos relativos ao subprograma de modo a simular a sua execução. A visualização que irá ser gerada para este exemplo é apresentada na figura 4.20. As duas regras de reescrita para efectuar estas acções no nodo de invocação estão esquematizadas na figura 4.15.

Após ter sido reescrita, a subárvore relativa ao subprograma, desaparecerá e será actualizada a tabela de identificadores global com a eventual alteração de variáveis globais. Uma variável que surge no subprograma mas não consta da tabela local é com certeza uma variável global. O valor do subprograma é posto no nodo `CALLPROG`. É de notar que as declarações das variáveis não terão imagem na árvore de execução. A regra de reescrita para retorno de subprogramas está esquematizadas na figura 4.16.

As regras de reescrita usadas para simular o funcionamento de subprogramas podem, então, ser especificadas da seguinte forma:

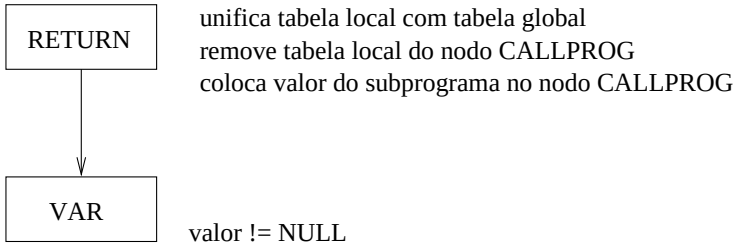

Figura 4.16: Regra de reescrita para o retorno de subprogramas

```
rew_rule(pcall1) = <a: callproc,b: lst>,
                   (getValue(a)=NULL),
                   <pcall2: a: callproc,b: lst,c: procdef>,
                   {end=getTableAddress(getName(a));
                    c=getNodeEnd(end);
                    setNodes(a,b,c);
                    setTabId(c,createTab());
                    setTabId(c,init(getTabId(c),getTabId(a),b)}

rew_rule(pret) = <r: return,e: exp>,
                 (getValue(e)!=NULL),
                 <pret:r: return,e: exp>,
                 {putTableId(getTabId(r),getName(r),getType(e),RETURN,
                             getValue(e),getEndNode(r),OUT)}

rew_rule(pcall2) = <a: callproc,b: lst,c: procdef>,
                   (getTableVal(getTabId(c),_,val,RETURN),
                    val!=NULL),
                   <pcall1: a: callproc,b: lst>,
                   {sincronize(getTabId(c),getTabId(a));
                    setTabId(c,NULL);
                    setValue(a,val)}
```

As regras `pcall1` e `pcall2` implicam reescritas sintácticas (introdução do nodo `PROCDEF` e remoção do nodo `PROCDEF` repectivamente). A árvore de execução ficará mais reduzida quando as invocações das funções já tiverem os valores calculados. Para o exemplo apresentado e após a execução do subprograma, a árvore de execução ficaria como na figura 4.17. É claro que as visualizações finais da animação já não

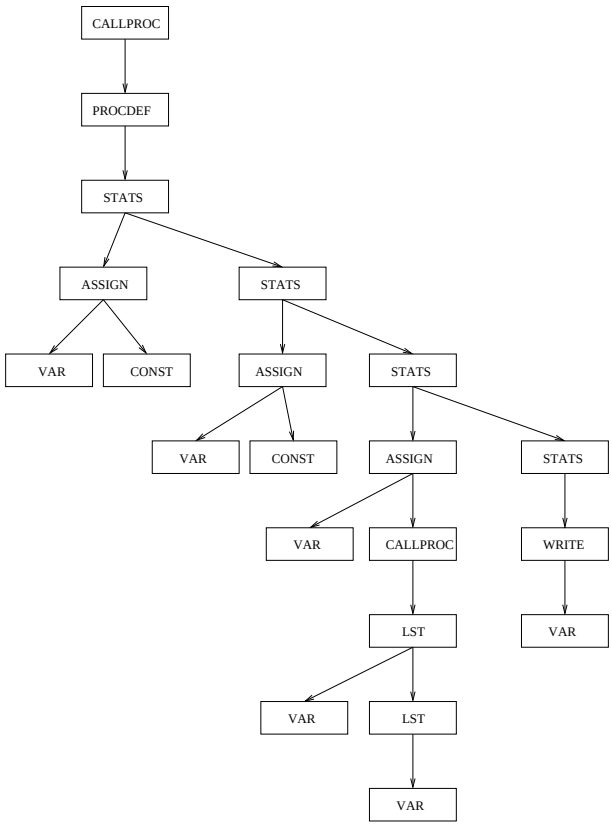

Figura 4.17: Árvore de execução com subprogramas calculados

conterão a visualização dos subprogramas e um nodo do tipo `CALLPROC` será visualizado como uma variável de retorno.

Quando o programa principal for todo reescrito, a árvore de execução volta a ter a forma da figura 4.13, embora o valor do nodo `CALLPROC` será agora diferente de nulo.

Tal como já foi dito, no final da execução de cada subprograma será necessário actualizar os valores da tabela principal com os valores retornados do subprograma: os parâmetros de saída; variáveis globais que eventualmente tenham sido alteradas; o valor retornado especificamente pelo subprograma. Tudo isto deve ser feito através de acções associadas ao nodo `RETURN` ou quando toda a subárvore do nodo `PROCDEF` estiver reescrita.

Relativamente às regras de visualização, é neste ponto acrescentada uma nova regra que pretende definir o desenho relativo a uma instrução de retorno, que pode ser vista na figura 4.18. As regras relativas aos nodos `CALLPROC` e `PROCDEF` são esquematizadas na figura 4.19. Relativamente à visualização do nodo `LST` já foi especificada na secção 4.7.3.

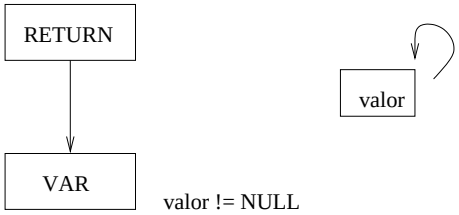

Figura 4.18: Regras de visualização para subprogramas (nodo RETURN)

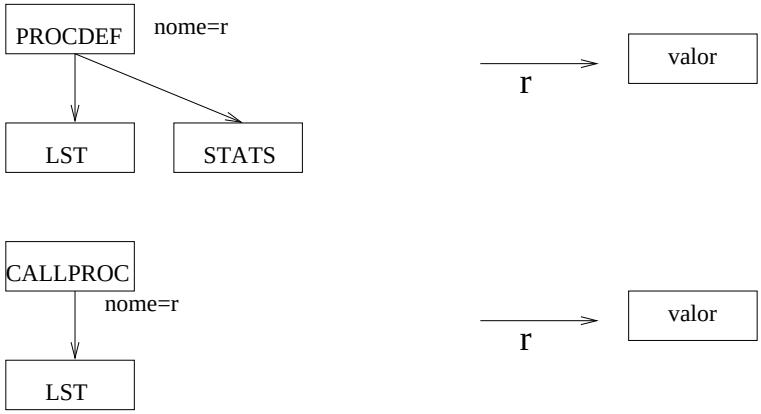

Figura 4.19: Regras de visualização para subprogramas

Aplicando estas regras, a visualização gerada aquando da invocação do subprograma do exemplo 4.8.1 (apresentado no início da secção), surge na figura 4.20. A visualização apresentada mostra a invocação do subprograma `soma()` (dentro de um rectângulo) e a passagem de parâmetros das variáveis `a` e `b` para `x` e `y`. O resultado desta invocação é depois guardado na variável `z`.

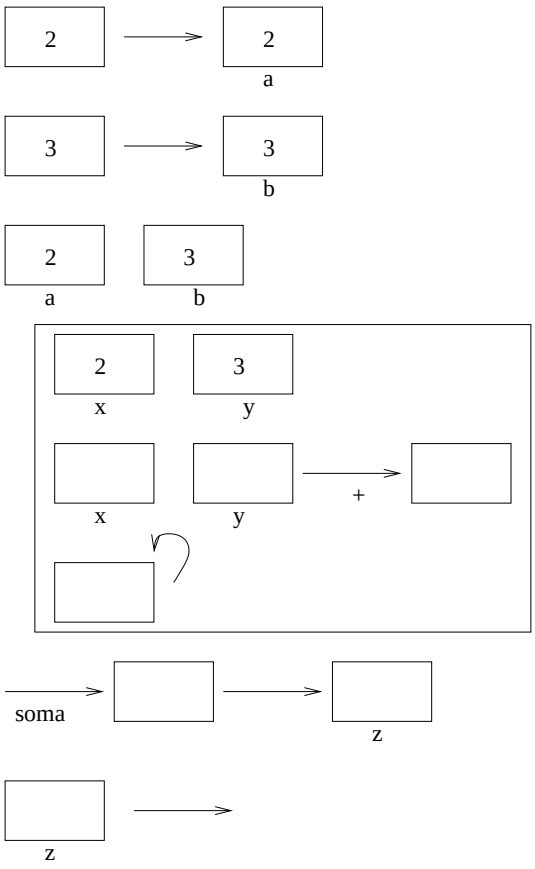

Figura 4.20: Visualização de subprogramas

4.9 Tratamento de variáveis estruturadas

As variáveis do tipo estruturado, representadas pelas últimas produções da gramática abstracta obrigam à especificação de operações para todos os tipos de variáveis. As regras de reescrita terão que ser mais abrangentes invocando essas operações tendo em conta os tipos das variáveis. Assim, as regras de reescrita deverão ser revistas e formalizadas da seguinte forma:

```
rew_rule(popr) = <op1: reloper,a: exp,b: exp>,
                 ( ),
                 <popr:op2: reloper,a: exp,b: exp>,
                 {setValue(op2,exec(getName(op1),getValue(a),
                           getType(a),getValue(b),getType(b)))}

rew_rule(pop) = <op1: oper,a: exp,b: exp>,
                ( ),
```

```
                    <pop:op2: oper,a: exp,b: exp>,
                    {setValue(op2,exec(getName(op1),getValue(a),
                              getType(a),getValue(b),getType(b)))}

rew_rule(passign) = <at: assign,a1: var,b: exp>,
                    ( ),
                    <passign: at: assign,a2: var,b: exp>,
                    {setName(a2,getName(a1));
                     setValue(a2,getValue(b),getType(b),getType(a1))}

rew_rule(pread) = <r: read,a: var>,
                  (getValue(a)=null),
                  <pread:r: read,a: var>,
                  {setValue(a,read(),getType(a))}

rew_rule(pwrite) = <w: write,a: var>,
                   (getValue(a)!=null),
                   <pwrite:w: write,a: var>,
                   {write(getValue(a),getType(a))}
```

O registo na tabela de identificadores das variáveis do tipo estruturado processa-se da mesma forma das variáveis do tipo simples. Mas as variáveis do tipo apontador obrigam a que seja criada mais uma linha na tabela para o seu apontado.

A animação de uma estrutura de dados consiste em desenhá-la sempre que há uma alteração de algum valor. A primitiva de desenho `varShape` terá que contemplar o desenho dessas estruturas com os respectivos valores, verificando previamente qual o tipo da variável.

A preparação do sistema para todo o tipo de variáveis implica apenas acrescentar algumas novas regras de reescrita e alterar a função de desenho `varShape`. Este problema não foi enquadrado nos objectivos principais do desenvolvimento do protótipo do sistema.

4.10 Exemplos de visualizações possíveis

Apresenta-se, nesta secção três exemplos de programas: um primeiro escrito numa linguagem de descrição de máquinas de estados (TSC); um segundo muito simples, escrito em Pascal e um terceiro, também escrito em Pascal mas com chamada de procedimentos. Para cada um dos exemplos é apresentada a DAST

e a visualização gerada.

O primeiro exemplo é mostrado com o objectivo de apresentar uma linguagem cujos programas são declarativos, ou seja, não possuem um caracter dinâmico que justifique a sua animação. Tomando como primeiro exemplo o seguinte programa:

Exemplo 4.10.1 (Programa escrito em TSC)

```
problem  SIMPLES
statemachine:  Sensor
states:  A,B
start:  init  ->  A
transitions  from  A:
e1 / e2 + incc  ->  B
transitions  from  B:
e1[count] / e1 +decc  ->  A;
e2[count] / e1 +decc  ->  B;
e1 / + incc  ->  B
end  of  description  SIMPLES
```

Pretende-se descrever o funcionamento de uma máquina chamada `Sensor` que tem dois estados de funcionamento (`A` e `B`), descrevendo as transições possíveis e respectivas condicionantes. Associando o conceito de transição a um nodo `RELOPER`, a árvore que foi gerada está representada na figura 4.21.

A informação representada na árvore não tem um caracter dinâmico ou, pelo menos, não contem nenhuma primitiva que despolete um processo de execução. Neste caso, uma visualização da árvore gerada numa primeira travessia constitui o resultado obtido pelo sistema Alma. Neste exemplo não se justifica a aplicação de regras de reescrita nem a construção da árvore de execução; são apenas aplicadas regras de visualização.

A visualização que se pretende que o sistema ALMA gere a partir da árvore está esquematizada na figura 4.22. Nesta visualização, para cada tipo de transição de estados, é mostrado o estado inicial e final e a condição que deve ser verificada para que tal ocorra.

Em oposição ao primeiro, o segundo exemplo apresenta um programa de caracter dinâmico que usa conceitos de leitura de variáveis, instruções condicionais, operações aritméticas e relacionais e atribuições. Neste caso, o *front-end* desta linguagens fará com que a DAST represente todos estes conceitos e as regras do sistema Alma conceder-lhe-ão a sua dinâmica. Considerando então o seguinte programa em Pascal:

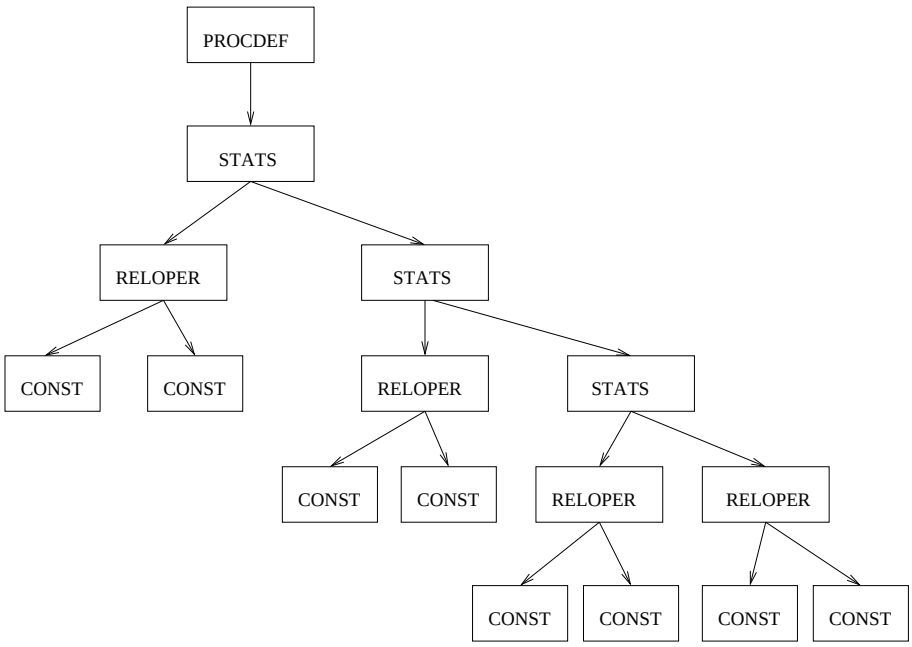

Figura 4.21: DAST gerada pelo *front-end* para o exemplo 4.10.1

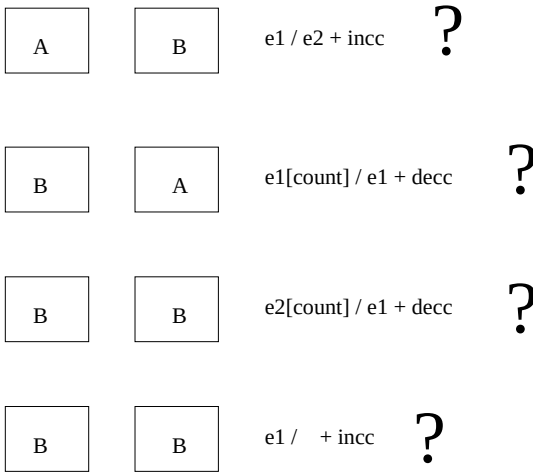

Figura 4.22: Visualização criada para o exemplo 4.10.1

Exemplo 4.10.2 (Excerto de um programa em Pascal)

```
PROGRAM resto;
VAR a,b :INTEGER;
BEGIN
read(a);
read(b);
```

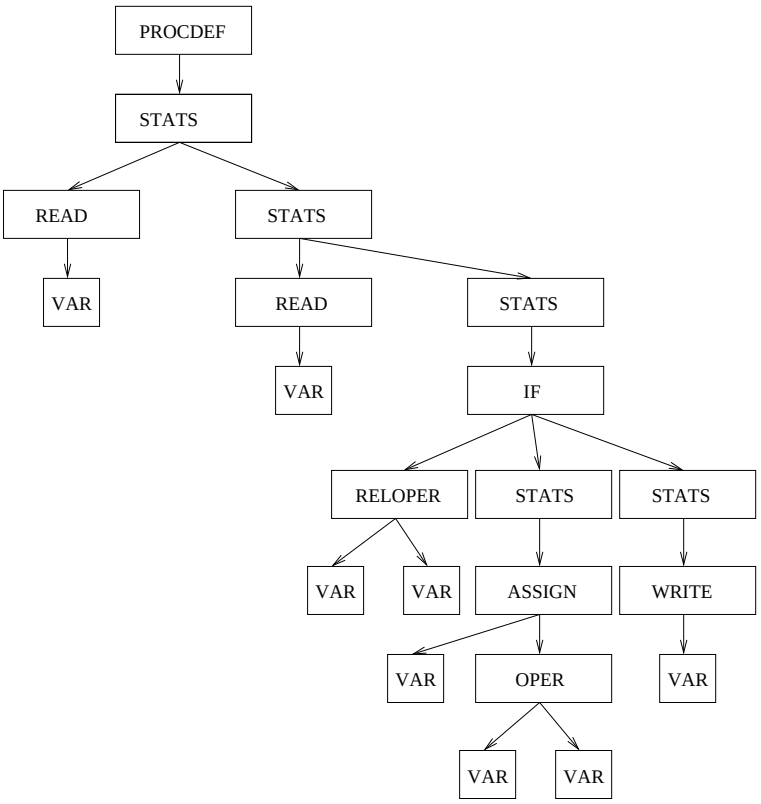

Figura 4.23: DAST gerada pelo *front-end* para o exemplo 4.10.2

```
if(a>b) then a:=a-b
        else write(b);
END.
```

A DAST correspondente é apresentada na figura 4.23. As regras de reescrita sincronizadas com as regras de visualização produzirão a animação. Algumas visualizações pertencentes a essa animação são apresentadas na figura 4.24. O símbolo `?` representa uma condição e o símbolo # uma instrução condicional. Os desenhos que surgem após este símbolo e que estão indentados fazem parte do corpo da instrução condicional. Neste caso, trata-se de uma operação de subtracção seguida de atribuição e uma instrução de escrita. Logo que as variáveis `a` e `b` são lidas a condição é calculada e as instruções associadas à primeira ou segunda parte da estrutura condicional desaparecem se a condição é falsa ou verdadeira respectivamente.

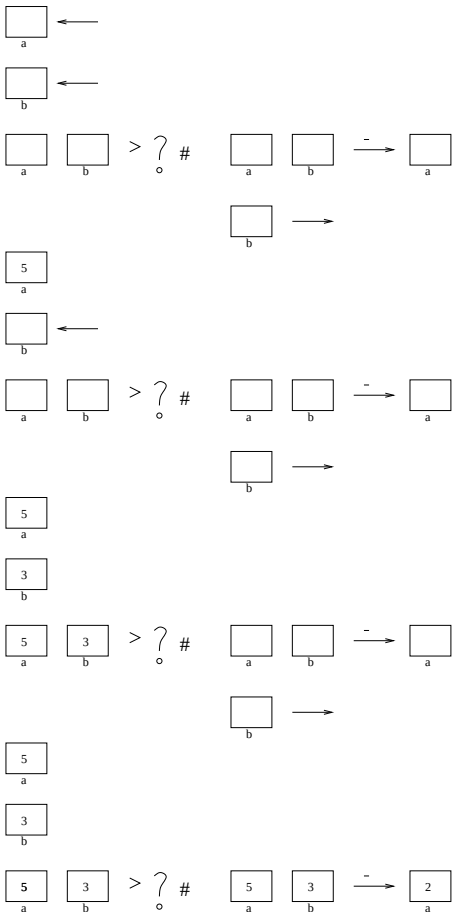

Figura 4.24: Alguns passos da animação do exemplo 4.10.2

O terceiro exemplo, mais completo, pretende ilustrar a animação de subprogramas. O programa a ser analisado será:

Exemplo 4.10.3 (Programa em Pascal com subprogramas)

```
PROGRAM parimpar;
VAR a,b: INTEGER;

FUNCTION par(x: integer):boolean;
VAR y:integer;
BEGIN
   y:=x MOD 2;
   if (y=0) then par:=true
            else par:=false;
END;
```

```
BEGIN
read(a);
read(b);
while(a>b) do
        begin
           if (par(a)) then write('par');
           a:=a-b;
        end;
END.
```

A DAST relativa a este exemplo será apresentada na figura 4.25.

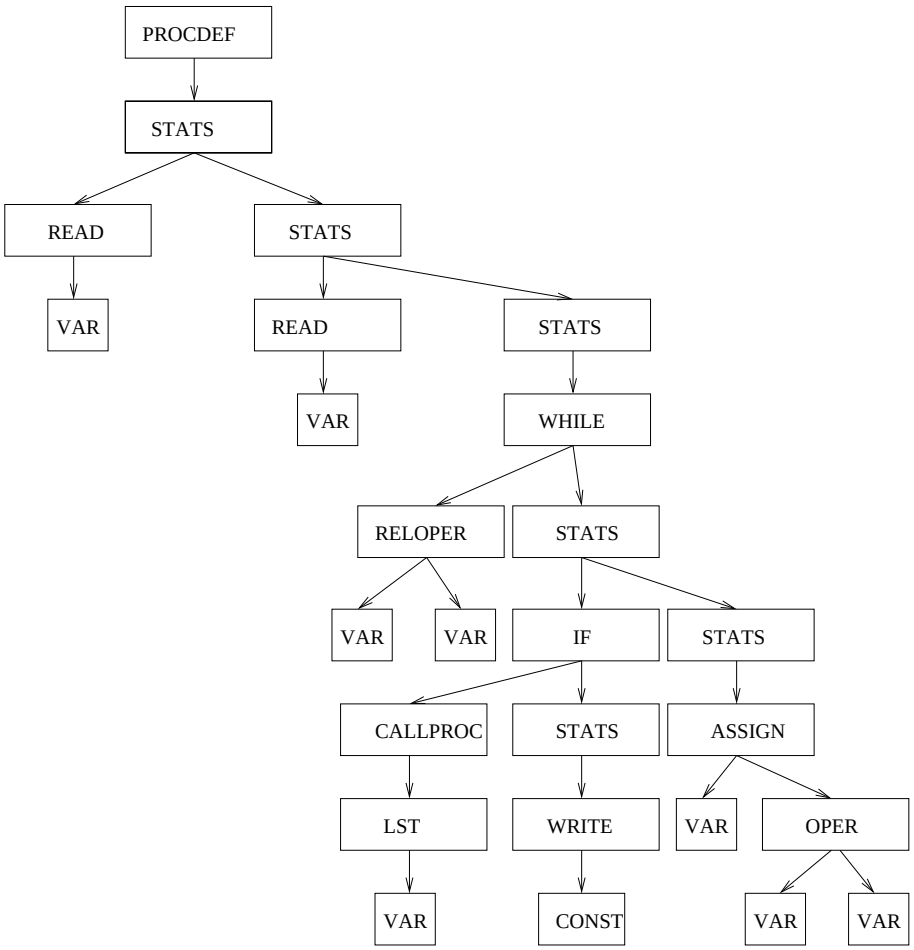

Figura 4.25: DAST gerada pelo *front-end* para exemplo 4.10.3

Alguns passos da visualização deste exemplo serão apresentados nas figuras 4.26 e 4.27. O símbolo `@` representa uma instrução repetitiva. Todos os desenhos indentados a seguir a este símbolo fazem parte do corpo da instrução. A visualização do subprograma surge dentro de um rectângulo, havendo uma passagem de parâmetro da variável `a` para a variável `x`. Quando a condição do ciclo se torna falsa as instruções do corpo da estrutura repetitiva desaparecem.Cada uma das figuras representa dois passos da animação gerada pelo Alma para este exemplo.

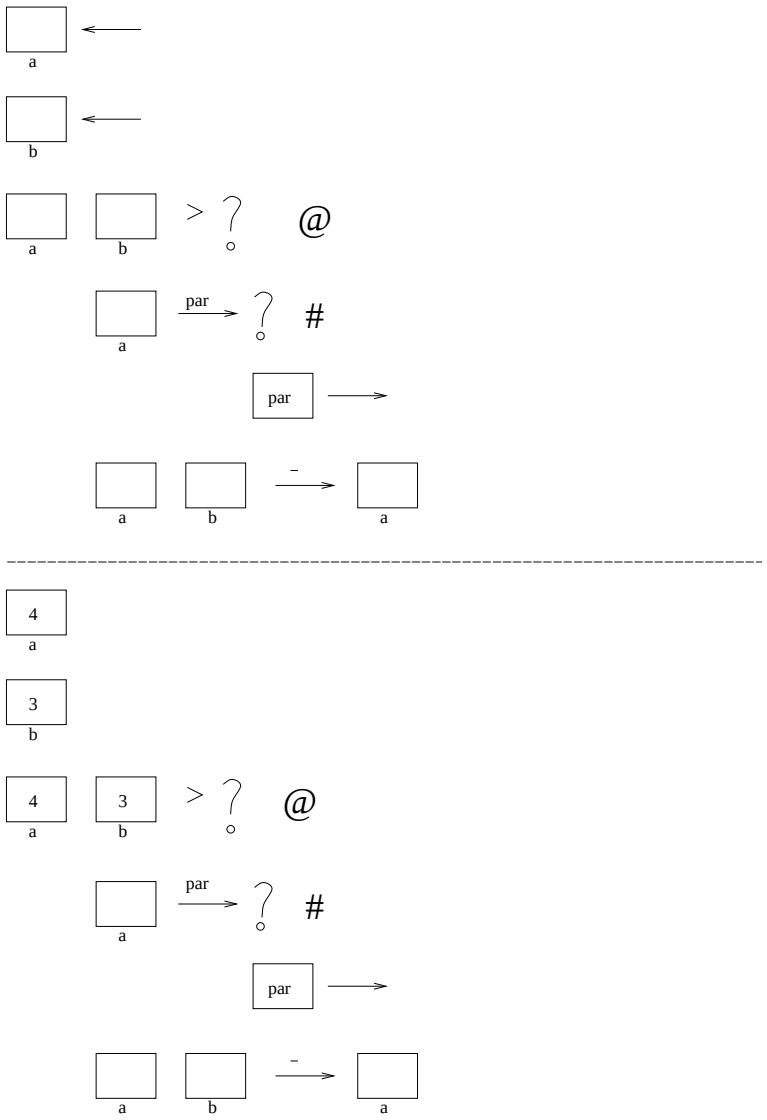

Figura 4.26: Animação do exemplo 4.10.3

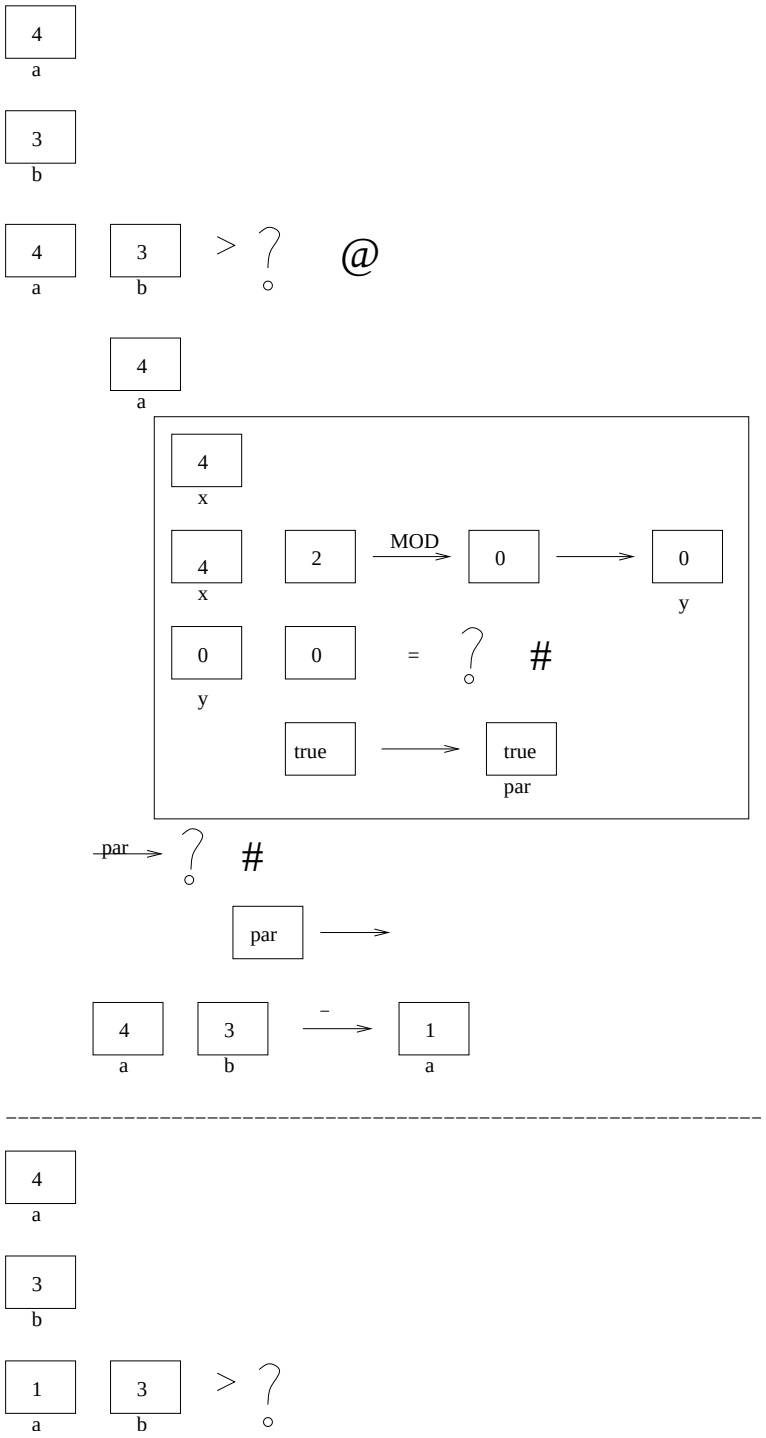

Figura 4.27: Animação do exemplo 4.10.3

Capítulo 5

Desenvolvimento do Sistema

Este capítulo pretende apresentar de forma sucinta, mas organizada, alguns pontos da implementação do sistema Alma. Nas próximas secções irá falar-se do uso do sistema LISA na construção de alguns *front-end*'s do sistema, assim como no reaproveitamento de classes na implementação do *back-end*, conforme foi apresentado em [HVML02]. O desenvolvimento deste último foi fortemente guiado pelas especificações discutidas no capítulo anterior, nomeadamente no que diz respeito à implementação da tabela de identificadores, construção dos nodos da DAST, sua reescrita e visualização. Por último, é dada uma nova perspectiva da arquitectura do sistema Alma especificando algumas das tarefas à custa de linguagens para domínios específicos (DSL's) reduzidas (simples e *curtas*) e mostrando como é que essas tarefas se realizam recorrendo a máquinas virtuais que as interpretam e constroem a animação.

5.1 Estratégia de implementação

De acordo com o diagrama de blocos apresentado na figura 5.1, o sistema baseia o seu funcionamento na Árvore de Sintaxe Abstracta Decorada e na Tabela de Identificadores. A figura 5.1 representa a arquitectura do sistema, mostrando como a representação interna se relaciona com o *front-end* e o *back-end*. Assim, é necessário identificar a informação que será armazenada nessa estrutura intermédia e de que forma esta deve ser usada para criar a animação. A implementação do sistema Alma requer então, a definição exacta dos tipos de nodos da DAST, atributos a associar aos nodos, travessias e formas de consulta à Tabela de Identificadores para gerar as animações pretendidas.
Após todas estas decisões, o desenvolvimento do sistema será dividido nas seguintes fases:

- Criação de um *front-end* para uma determinada linguagem, a partir da respectiva gramática (para

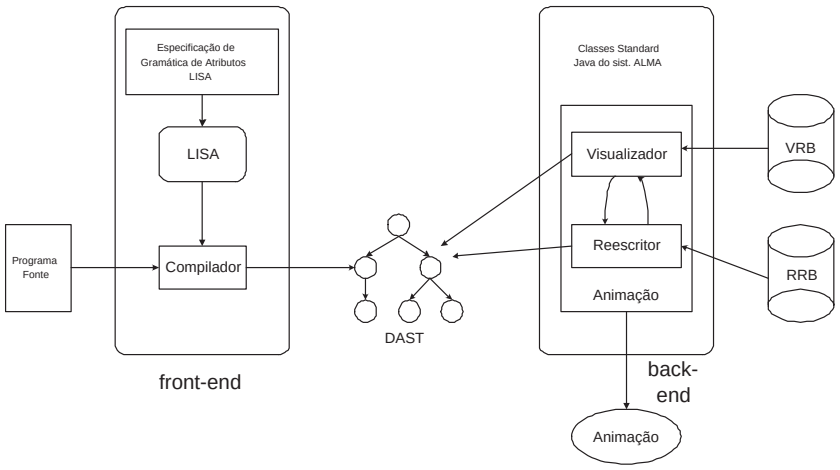

Figura 5.1: Arquitectura do sistema Alma

diferentes linguagens, o *front-end* deverá ser gerado de novo de acordo com as suas gramáticas).

```
Front-End :  Seq(caracteres) -> DAST
```

- Criação de um *back-end* que, a partir da informação extraída pelo *front-end*, realize a animação pretendida (o *back-end* será independente da linguagem fonte). Este *Back-end* será implementado com base no algoritmo de um Tree-Walker Animator que ao visitar cada nodo da DAST (segundo um percurso pré-definido) aplicará as operações de visualização, filtragem e transformação adequadas. Dividimos, assim, o *Back-end* em duas componentes principais, que se especificam a seguir:

```
Visualizador    : DAST -> desenho do programa
Reescritor      : DAST -> DAST
```

Considerando a DAST decorada com valores de atributos e figuras, a sua travessia (efectuada pelo visualizador) terá como resultado um desenho global do programa. Numa segunda etapa a DAST é reescrita de modo a que uma nova travessia gere um novo desenho.

5.2 Utilização do sistema LISA na geração dos *front-end*'s

Por diversas razões, das quais se vai falar nesta secção, decidiu-se utilizar o sistema LISA para a implementação do sistema Alma. Pretende-se também, nesta secção, mostrar como foi usado o sis-

tema LISA na construção dos *front-end*'s do Alma. Para tal, é explicada a forma de especificação de linguagens no LISA e como uma gramática pode ser estendida de forma a gerar o *front-end* adequado.

5.2.1 Escolha do sistema LISA

A escolha do gerador de compiladores LISA, como suporte para o desenvolvimento do novo sistema, baseou-se num conjunto de características especiais: tal como já foi dito na secção 2.6, é independente da plataforma utilizada (porque está escrito em Java); oferece a possibilidade de trabalhar num ambiente com interface gráfico (baseado em janelas) amigável; oferece um ambiente de desenvolvimento integrado, onde os utilizadores podem especificar, gerar analisadores, compilar e executar programas escritos numa nova linguagem. Tal como também já foi dito na secção 2.6, o sistema gera ainda apresentações visuais de diferentes estruturas associadas aos compiladores criados, tais como: autómatos finitos, gramática BNF, árvore de sintaxe, árvore semântica e grafo de dependências (ver apêndice E), assim como, visualizações dos processos de análise léxica, sintáctica e semântica. É de notar que, estas visualizações e animações dizem respeito à execução do compilador gerado e não do programa fonte que está a ser compilado. Por último, as especificações de uma nova linguagem são feitas com base em gramáticas de atributos que suportam herança múltipla, o que permite desenhar uma linguagem incrementalmente ou reutilizar especificações de outras linguagens.
O sistema LISA preenche os requisitos necessários à implementação desejada: contem uma interface com o utilizador muito eficiente e de fácil utilização que serve para especificar a linguagem fonte, gerar o compilador e submeter textos fonte ao compilador gerado; pode ser usado, com facilidade, para o desenvolvimento dos *front-end*'s necessários para o Alma; usa uma técnica de implementação (quer do gerador quer do compilador gerado) que permite uma reutilização expedita das suas classes na construção do *back-end* do Alma.

Analisadas as necessidades de implementação do Alma, é então, possível decidir que o sistema LISA irá ser usado e reutilizado em vários pontos da implementação:

- será usado para gerar automaticamente cada *front-end* que se queira adicionar ao sistema Alma
- os seus editores serão usados não só para especificar a nova linguagem (construção do *front-end*), mas também para editar o texto fonte a animar e visualizar a animação gerada

- as visualizações do autómato da análise léxica, da gramática concreta, da árvore de sintaxe com ou sem cálculo de atributos, embora relacionadas com o compilador gerado e não com o programa fonte, constituem motivo de interesse para o utilizador final do sistema Alma

- algumas classes Java usadas pelo sistema LISA na geração do compilador irão ser reutilizadas na construção e manipulação da DAST do Alma, e na implementação dos processos de animação

5.2.2 Especificação de linguagens no sistema LISA

Tal como já foi dito na secção anterior, LISA é um compilador de compiladores, ou seja, um sistema que automaticamente gera um compilador/interpretador a partir de uma especificação de linguagem baseada em gramáticas de atributos. A sintaxe e a semântica das especificações LISA, assim como algumas extensões como o uso de *templates* e de herança múltipla em gramáticas de atributos, são descritos em pormenor em [MLAZ00].

Os geradores que normalmente se usam não permitem desenvolvimento incremental de linguagens nem a reutilização de fragmentos de especificações. Este sistema aplica o conceito de herança, uma das principais características da programação orientada ao objecto, às gramáticas de atributos. Uma gramática de atributos é uma generalização das gramáticas independentes de contexto, onde cada símbolo tem associado um conjunto de atributos que contêm informação semântica.

A herança múltipla de gramáticas de atributos é uma organização estrutural onde uma nova gramática de atributos pode herdar as especificações semânticas de outra gramática, pode adicionar novas especificações e pode sobrepor apenas algumas especificações dessa gramática. No entanto, este novo conceito não ajuda quando as linguagens têm semânticas similares e sintaxes diferentes. Nessas situações são usados *templates*, sendo um *template*, no contexto do sistema LISA, uma abstracção polimórfica de uma regra semântica parametrizada com ocorrências de atributos que podem estar associados a produções com diferentes símbolos não-terminais e terminais. Desta forma, o desenhador apenas escreve novas especificações, reutiliza outras e pode criar os seus próprios *templates*.

A gramática da linguagem é especificada numa notação semelhante à das linguagens orientadas ao objecto. A gramática fica então dividida em várias gramáticas e, em cada uma delas, especifica-se os atributos (variáveis) e os cálculos associados a cada produção (métodos).

A partir dos vários módulos, o sistema LISA cria uma especificação monolítica que inclui as especificações de todas as linguagens e de todas as produções. Para ilustrar o estilo de uma gramática LISA, apresenta-se como exemplo a especificação da linguagem *Simple Expression Language with Assign-*

ments, SELA.

```
language SELA {
   lexicon
         {
                  Number      [0-9]+
                  Identifier  [a-z]+
                  Operator    \+ | :=
                  ignore      [\0x09\0x0A\0x0D\  ]+
         }
   attributes Hashtable *.inEnv, *.outEnv;
              int *.val;

   rule Start
         {     START ::= STMTS compute
                  {     STMTS.inEnv = new Hashtable();
                         START.outEnv = STMTS.outEnv;
                  };
         }
   rule Statements
         {     STMTS ::= STMT STMTS compute
                  {     STMT.inEnv = STMTS[0].inEnv;
                         STMTS[1].inEnv = STMT.outEnv;
                         STMTS[0].outEnv = STMTS[1].outEnv;
                  }
                | STMT  compute
                  {     STMT.inEnv = STMTS[0].inEnv;
                         STMTS[0].outEnv = STMT.outEnv;
                  };
         }
   rule Statement
         {     STMT ::= #Identifier \:= EXPR compute
                  {     EXPR.inEnv = STMT.inEnv;
                         STMT.outEnv = put(STMT.inEnv,
                         #Identifier.value(), EXPR.val);
                  };
```

```
        }
    rule Expression
        {     EXPR ::= EXPR + EXPR compute
              {     EXPR[2].inEnv = EXPR[0].inEnv;
                    EXPR[1].inEnv = EXPR[0].inEnv;
                    EXPR[0].val = EXPR[1].val + EXPR[2].val;
              };
        }
    rule Term1
        {     EXPR ::= #Number compute
              {     EXPR.val = Integer.valueOf(
                                 #Number.value()).intValue();
              };
        }
    rule Term2
        {     EXPR ::= #Identifier compute
              {     EXPR.val = ((Integer)EXPR.inEnv.get(
                                #Identifier.value())).intValue();
              };
        }
    method Environment
        {
              import java.util.*;
              public Hashtable put(Hashtable env, String name, int val)
                {
                    env = (Hashtable)env.clone();
                    env.put(name, new Integer(val));
                    return env;
                }
        }
}
```

Como se pode constatar, as especificações LISA são concisas, claras e directas. Do exemplo pode concluir-se que a especificação de uma linguagem começa pela descrição dos seus símbolos terminais à qual se segue a associação de atributos (devidamente tipados) aos símbolos. A seguir, a especificação

é organizada em módulos (`rule`'s) correspondendo cada um a um símbolo não-terminal; o módulo contém as várias produções gramaticais alternativas e para cada uma inclui o conjunto de todas as regras de cálculo de atributos herdados ou sintetizados associados aos símbolos da produção em causa. A especificação termina com um bloco operacional onde se pode definir métodos Java que tenham sido usados nas regras de cálculo de atributos. Note-se que em LISA a especificação de condições de contexto (impor restrições semânticas aos atributos envolvidos em cada produção) e as regras de tradução são escritas na forma, já referida, de regras de cálculo de atributos.A partir destas descrições, LISA gera automaticamente um compilador/interpretador para SELA.

A geração automática de ferramentas é possível quando estas podem ser construídas com base numa parte fixa e numa parte variável, sendo a parte variável (dependente da linguagem fonte) derivável sistematicamente das especificações de cada linguagem concreta. Esta parte variável tem uma representação interna que pode ser atravessada pelos algoritmos da parte fixa.

5.2.3 Extensão da gramática para construção dos *front-end*'s

Podemos deduzir das secções anteriores que ferramentas como animadores e visualizadores de programas podem ser derivados da especificação formal das linguagens. O objectivo desta família de sistemas é ajudar o programador a inspeccionar o fluxo de controlo e de dados do programa fonte (visão estática dos algoritmos subjacentes aos programas) e compreender o seu comportamento (visão dinâmica da execução dos algoritmos). Este tipo de ferramenta pode ser obtida pela especificação de um animador genérico (um *back-end* independente da linguagem) que trabalha sobre o resultado de um *front-end* construído com base numa extensão à gramática de atributos LISA que especifica a linguagem a ser analisada.
A extensão à gramática de atributos apenas define a forma como uma frase de entrada deve ser convertida para a representação interna do animador (DAST). Considerando o exemplo já apresentado (a especificação LISA da linguagem SELA) surge agora uma extensão a essa gramática, para construir a DAST referente a essa linguagem.

```
import  "AlmaLib.lisa";
language  AlmaSELA  extends SELA,  AlmaBase  {
   rule extends  Start
         {
                START ::= STMTS compute
                  {     ALMA_ROOT<START,STMTS>
```

```
                };
        }
    rule  extends Statements
        {     STMTS ::= STMT STMTS compute
                {     ALMA_STATS<STMTS, STMT,  STMTS[1]>
                }
               | STMT  compute
                {     ALMA_IDENT<STMTS, STMT>
                };
        }
    rule  extends Statement
        {     STMT ::= #Identifier := EXPR compute
                {
                       ALMA_ASSIGN<STMT,ALMA_VAR(#Identifier), EXPR>
                };
        }
    rule extends  Expression
        {     EXPR ::= EXPR + EXPR compute
                {
                       ALMA_OPER<EXPR[0],EXPR[1], EXPR[2], "+">
                };
        }
    rule  extends Term1
        {     EXPR ::= #Number compute
                {
                       ALMA_CONST<EXPR,#Number>
                };
        }
    rule extends  Term2
        {     EXPR ::= #Identifier compute
                {
                       ALMA_VAR<EXPR,#Identifier>
                };
        }
}
```

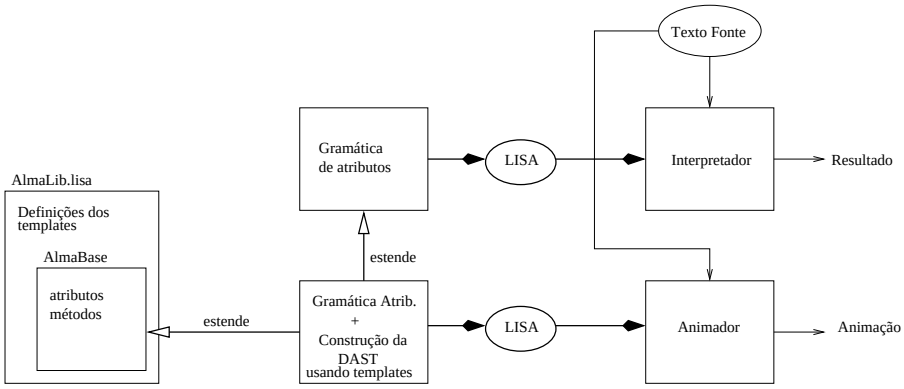

Figura 5.2: Geração do animador a partir de uma especificação LISA

Esta extensão mostra o uso de *templates* (importados do ficheiro `AlmaLib.lisa`[1] e empregues na escrita das regras de cálculo dos atributos) e de herança múltipla de gramáticas de atributos (além das regras explicitadas, cada produção conterá ainda as regras definidas na especificação da linguagem SELA (na subsecção 5.2.2)). A linguagem `AlmaBase` (definida também no ficheiro `AlmaLib.lisa`) consiste apenas na definição dos atributos usados para a construção da DAST pelos *templates* já referidos. A partir desta especificação é gerado um *parser* e um tradutor que converte cada texto de entrada numa representação abstracta usada pelo animador e comum a todas as linguagens fonte. Esse processador, ao qual se chama o *front-end* do animador é a componente do sistema que é dependente da linguagem. A parte fixa do sistema (a discutir na próxima secção) é constituída por algoritmos e algumas estruturas independentes da linguagem fonte, nomeadamente, uma base de regras de visualização e uma base de regras de reescrita.

Na figura 5.2 é dada uma visão da biblioteca geral do Alma, da especificação de um *front-end* concreto e da sua relação com um compilador / interpretador clássico gerado pelo sistema LISA.

Assim, considerando o seguinte programa fonte escrito em SELA:

Exemplo 5.2.1 (Programa de entrada)

```
a:=2+2+5
b:=a+3+a
c:=a+b+6
```

[1] O ficheiro `AlmaLib.lisa` consta no apêndice C

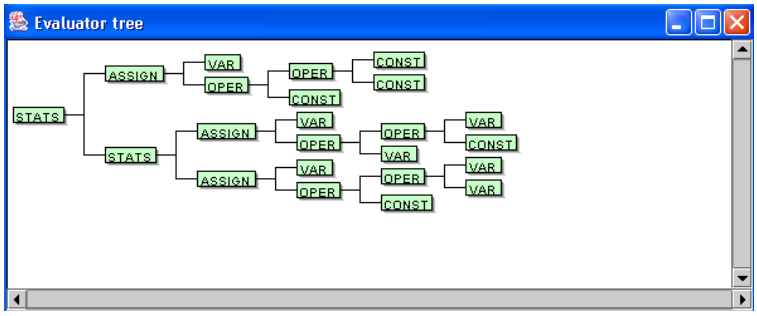

Figura 5.3: DAST gerada pelo *front-end* para o exemplo 5.2.1

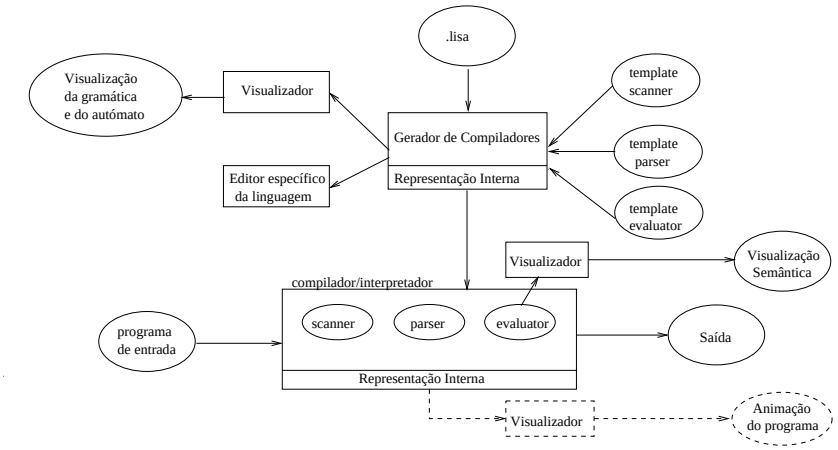

Figura 5.4: Arquitectura do sistema LISA

a DAST gerada pelo *front-end* pode ser vista na figura 5.3.

5.3 Reutilização do sistema LISA no BE do sistema Alma

LISA em si e os compiladores por ele gerados são implementados em Java, seguindo uma abordagem orientada ao objecto. Na figura 5.4 obtém-se uma visão geral da arquitectura do sistema LISA integrado com todas as ferramentas incluídas no gerador. Assim, é muito fácil identificar e compreender as estruturas e as funções usadas pelo sistema LISA para processar a gramática de atributos ou os programas fonte, funções e estruturas que estão encapsulados em classes como atributos ou métodos.

Essa especificação LISA permite gerar o compilador com base *templates* e ficheiros .java que contêm métodos java para manusear a representação interna. Assim, são gerados os ficheiros: `scanner.java`,

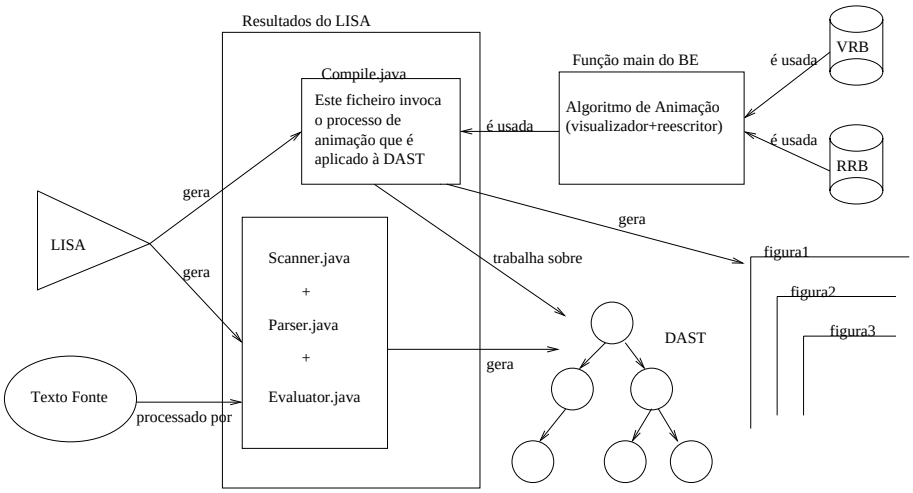

Figura 5.5: Ligação entre o sistema LISA e o sistema Alma

`parser.java`, `evaluator.java` e `compiler.java` que constituem o compilador pretendido. Terminar a geração do compilador corresponde a compilar todos estes ficheiros java. Executar o compilador significa executar (em Java) os ficheiros `.class` resultantes da fase anterior.

A codificação das estruturas de dados e dos algoritmos necessários para implementar o sistema Alma segue a mesma abordagem orientada aos objectos e vai reutilizar, por decisão de desenho tomada neste projecto, algumas das referidas classes.

Uma visão geral do sistema Alma é apresentada na figura 5.5. Para construir a DAST – que é o *output* do *front-end* do Alma (gerado pelo LISA) e o *input* do seu *back-end* do Alma, o processador da especificação da gramática de atributos deverá invocar alguns métodos específicos (fornecidos pela biblioteca standard do Alma) para criar um novo nodo da árvore para cada símbolo da linguagem abstracta da DAST e ir coleccionando os nodos que constituem a árvore. Para implementar esses métodos, reutiliza-se classes Java como: `CTreeNode`, `CSyntaxTree` e `CParserSymbol`, usadas pelo LISA para criar a sua própria representação interna.

Como consequência imediata: todas as facilidades produzidas pelo ambiente LISA para manipular essas árvores ficam também disponíveis para processar a DAST. Essas classes serão usadas novamente para construir as regras (de visualização e de reescrita) necessárias para a implementação dos algoritmos de visualização e animação do *back-end*. Como já foi dito em secções anteriores todas as regras de visualização e de reescrita são definidas em termos de *tree-patterns*. O *back-end* do Alma é também uma

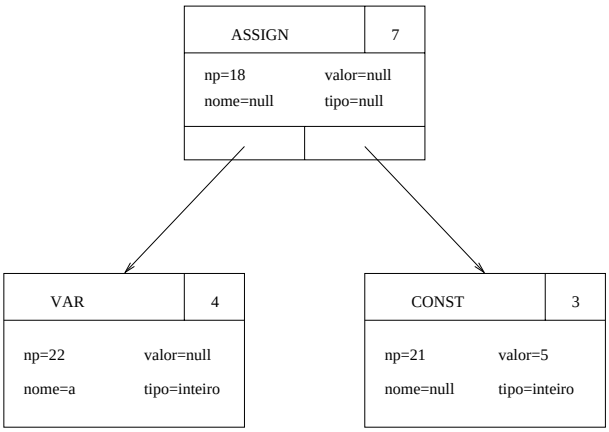

Figura 5.6: Estrutura dos nodos da DAST

classe Java, especialmente construída para este propósito, que usa as estruturas de dados e implementa os algoritmos directamente. Também para codificar esta classe se manteve uma abordagem orientada ao objecto seguida no desenvolvimento do sistema LISA.

A classe *main* do sistema Alma contem os métodos necessários para invocar e sincronizar o *front-end* do animador e as funções do *back-end* e foi implementada reutilizando o ficheiro `compile.java` gerado pelo LISA.

5.4 Alguns detalhes de implementação

Nesta secção pretende-se explorar um pouco mais a filosofia adoptada para implementar o *back-end* do sistema Alma. Embora tenham sido reutilizadas algumas classes do sistema LISA, a implementação desta parte ocorreu de forma independente. O *back-end* é composto por um conjunto de ficheiros Java que trabalham sobre uma estrutura do tipo árvore resultante do *front-end*.

5.4.1 A DAST gerada

Como resultado do *front-end* obtemos uma estrutura em árvore n-ária, cujos nodos contêm informação relativa ao símbolo que representam, um conjunto de atributos e o número da produção da gramática abstracta do Alma a que diz respeito. Considere como exemplo a subárvore representativa de uma atribuição que é apresentada na figura 5.6.

Os atributos `nome`, `valor` e `tipo` são relativos ao elemento do programa que o símbolo nesse nodo da árvore está a representar. Se se tratar de uma variável faz sentido que os atributos `nome` e `tipo` estejam

definidos desde o início da execução do programa, assim como, o valor das constantes. Os restantes atributos serão preenchidos à medida que o cálculo vai evoluindo na árvore por aplicação das regras de reescrita. Todas as regras, quer de reescrita, quer de visualização, são aplicadas consoante o número da produção que representam, não sendo portanto necessário verificar se os nodos da regra e os nodos da DAST são do mesmo tipo, para que a regra seja aplicada.

No apêndice C é apresentado o código das funções de construção dos nodos da DAST.

TABELA DE SÍMBOLOS DA DAST		
Nome do Símbolo	Código do Símbolo	Número da Produção
CONST	3	21
VAR	4	22
READ	5	19
WRITE	6	20
EXP	30	30, 31, 32, 33, 34
CALLPROC	2	3
STATS	1	1, 2
ASSIGN	7	18
OPER	8	17
RELOPER	9	15, 16
IF	10	23, 25
WHILE	11	24
LST	35	35, 36
RETURN	37	37
PROCDEF	38	38

Tabela 5.1: Códigos e números de produção dos símbolos da gramática abstracta do Alma

O código de um símbolo serve para identificá-lo para além do nome. O número de produção serve para identificar os seus descendentes porque o mesmo símbolo poderá ter diferentes hipóteses de derivação dependendo da produção em causa. O número da produção identifica claramente a subárvore da DAST que devemos gerar. Os nodos `CALLPROC` e `PROCDEF` representam conceitos relacionados com sub-programas; a semântica dos subprogramas obriga à criação de *ambientes locais*, onde existe uma tabela de identificadores própria. Havendo várias tabelas numa só árvore, surge a necessidade de indicar para

cada nodo qual a tabela a usar. Assim, um quarto atributo de cada nodo indica a tabela de identificadores (global se o nodo corresponde ao programa principal ou local se se trata de um subprograma).

5.4.2 Algoritmo principal do *back-end*

O algoritmo principal do sistema Alma é responsável por receber a DAST construída pelo *front-end* e de lhe aplicar um algoritmo de animação. Esse algoritmo consiste em invocar alternadamente um algoritmo de visualização e um algoritmo de reescrita de modo a gerar uma sequência temporal de visualizações, que constitui a animação pretendida (ver apêndice D).
O algoritmo começa por carregar as duas bases de regras: de visualização e de reescrita. Em seguida aplica algumas regras de reescrita que transformam a árvore de forma a representar a semântica operacional do programa, ou seja, constrói a árvore de execução. Ao nodo raiz será associada uma tabela de identificadores.
Antes de iniciar o processo de visualização, o algoritmo *decora* a árvore com algumas informações importantes como sendo: o nível de aninhamento das instruções e número de instruções agrupadas em ciclos e estruturas condicionais.
Inicia-se, neste ponto, um processo cíclico de visualização seguida de reescrita, produzindo diversas imagens que constituem a animação. É também criado um vector onde fica registada a regra de reescrita aplicada em cada travessia. Quando este vector é nulo, todo o processo de construção da animação pára. Não poder aplicar mais nenhuma regra significa que a árvore não vai ser modificada e, sendo assim, uma nova visualização será desnecessária.

Sempre que a função de visualização é invocada, verifica-se antecipadamente em cada nodo a sua visibilidade, ou seja, quando um nodo tem uma etiqueta “SKIP” significa que não irá ser visualizado. A função de visualização efectua uma travessia *post-order* tentando aplicar em todos os nodos alguma regra de visualização. Aplicar esta regra significa guardar numa estrutura especial os desenhos, valores e etiquetas que farão parte da visualização gerada.
O algoritmo de reescrita efectua uma travessia *pre-order* tentando aplicar alguma regra de reescrita, verificando igualmente, em cada nodo, a sua visibilidade. Logo que uma regra é aplicada, o processo de reescrita finda. Depois de efectuado o cálculo está na altura de produzir uma nova visualização. O processo de animação entra então em ciclo onde em cada iteração a árvore é reescrita e visualizada. Numa animação *passo a passo*, uma reescrita implica sempre uma visualização mas poderá haver casos

em que isso não é oportuno. Existe então uma função booleana cujo valor varia consoante a frequência de amostragem.

Quando o ciclo terminar (a árvore não será mais reescrita), a estrutura especial de desenhos que foi preenchida ao longo de todo o processo será usada para desenhar todas as visualizações da animação num editor do sistema LISA.

É de notar que o algoritmo de reescrita tem que alterar a sua travessia na árvore quando encontra nodos pertencentes a uma estrutura cíclica. Ou seja, quando encontra um nodo com a etiqueta "REP" força uma nova travessia na subárvore encabeçada por esse nodo. Os atributos serão recalculados o número de vezes necessário para que a execução do ciclo esteja completa. Para isso, e antes de cada nova travessia é verificada o valor da condição do ciclo e é feito o *reset* do cálculo das operações para que estas sejam novamente feitas mas com valores actualizados das variáveis, vindos da tabela de identificadores.

As estruturas condicionais são implementadas com base no uso da etiqueta "SKIP" posta nos nodos em que a condição é falsa.

5.4.3 Bases de regras

Fazem parte do sistema Alma, um conjunto de regras de reescrita e um conjunto de regras de visualização. De uma forma geral, cada regra está dividida em duas partes: um conjunto de nodos (modelo de árvore [2]) com atributos especiais (expressões que constituem as condições de aplicação da regra e que são relativas ao nodo em causa - `expressions`) e um conjunto de acções (alterações dos atributos que também são especificadas em cada nodo - `actions`) ou um conjunto de desenhos (no caso das regras de visualização). O modelo de árvore que surge em cada regra é constituído por um nodo pai e 1,2 ou 3 nodos filhos. Um exemplo é apresentado na figura 5.7.

Para que esta regra (ilustrada na figura 5.7) seja aplicada, o atributo `valor` associado ao nodo `EXP` não pode ser nulo e o atributo `valor` associado ao nodo `VAR` deve ser nulo (indica que a operação de atribuição ainda não foi efectuada). Assim, aplicada a regra (se ambas as expressões condicionais associadas aos nodos `VAR` e `EXP` forem verdadeiras), o valor de `EXP` é passado para o nodo `VAR` e guardado na Tabela de Identificadores. O nodo `ASSIGN` não tem expressões nem acções associadas.

Este modelo de árvore terá que efectuar com sucesso a concordância com alguma subárvore da DAST. Por sua vez, esses nodos da DAST deverão verificar as condições que constam em cada modelo de nodo respectivo. Se tal acontecer as acções que constam nos mesmos nodos serão executadas (no caso das

[2] do inglês *tree-pattern*

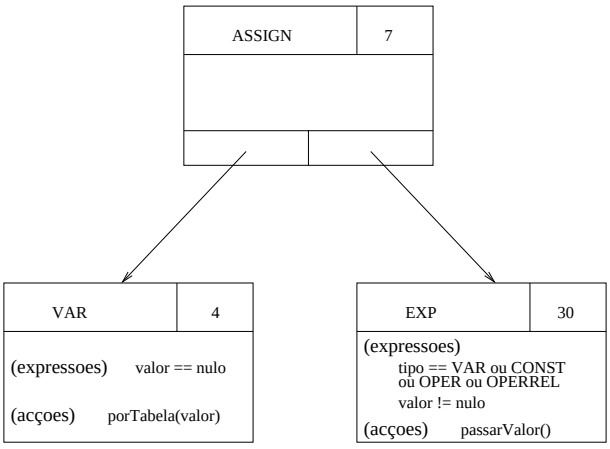

Figura 5.7: Estrutura dos modelos de nodos que constam nas regras

NOME nodo pai	VALOR nodo pai	NOME nodo filho 1	VALOR nodo filho1	NOME nodo filho 2	VALOR nodo filho 2	NOME nodo filho 3	VALOR nodo filho 3

Figura 5.8: Estrutura auxiliar das regras

regras de reescrita). No caso das regras de visualização nenhuma acção consta nos modelos de nodos mas um conjunto de desenhos irá representar estes nodos na animação.

Algumas expressões não têm por objectivo condicionar o uso da regra mas preparar os dados necessários para executar as acções. Assim, os nomes e valores de variáveis e operações que serão precisos para efectuar cálculos ou simples atribuições ou ainda para integrar a visualização gerada, serão guardados numa estrutura auxiliar. O visualizador e as acções de reescrita terão toda a informação disponível. Essa estrutura auxiliar está organizada como consta na figura 5.8.

Os desenhos são elementos gráficos como setas, figuras geométricas, traços ou outros símbolos acompanhados de valores ou etiquetas que representam nomes de variáveis e de operações. Os comandos de desenho que podem ser utilizados têm nomes como: `varShape` (variáveis); `operShape` (operações aritméticas); `operrelShape` (operadores relacionais e lógicos); `readShape` (leitura) ; `writeShape` (escrita); `assignShape` (atribuição); `whiteShape` (desenho em branco usado nas indentações). Note-se que a função `varShape` servirá para os mais variados tipos de variáveis.

Estes comandos adicionados à informação armazenada na estrutura auxiliar da figura 5.8, são guardados numa estrutura sequencial que irá ser interpretada pelo gerador de desenhos do sistema LISA.

As acções das regras de reescrita são especificadas usando comandos de reescrita tais como: `toread` (leitura de valores do teclado); `towrite` (escrita de valores no ecrã); `putTable` (inserção de valores na tabela); `putNode` (inserção de valores no nodo da DAST); `writing` (escrita de valores na estrutura auxiliar). Existem dois comandos que preparam valores para que as acções possam ser executadas: `reading` (transfere valores dos nodos da DAST para a estrutura auxiliar); `getTable` (transfere valores da tabela de identificadores para a estrutura auxiliar).
Cada regra tem três momentos: o momento em que regra é carregada para o sistema, um outro em que se verifica a sua adequação e um terceiro onde a regra é efectivamente aplicada. Neste terceiro momento e numa regra de reescrita, é feito o cálculo das operações, a leitura e escrita de valores e a alteração dos atributos dos nodos da DAST.

5.4.4 Tabela de Identificadores

A Tabela de Identificadores conterá todas as variáveis, constantes e nomes de funções do programa cuja execução está a ser simulada. Para cada um destes é guardado: o nome; o tipo (inteiro,float,... ou tipo de dados estruturado); a classe (nome do símbolo que o representa na árvore); o valor; o endereço do nodo na DAST; e o tipo de parâmetro que o identificador pode representar (entrada, saída ou ambos). O campo nome é nulo no caso das constantes e o valor das variáveis pode ser introduzido quando se insere a variável pela primeira vez (`putTableId`) ou depois quando é calculada ou alterada, usando uma outra função (`putTableVal`).

Nome	Tipo	Classe	Valor	Endereço	Tparam
a	int	VAR	null	2	IN
b	int	VAR	3	6	IN
atrib	void	PROCDEF	null	12	OUT
soma	int	PROCDEF	null	20	OUT

Tabela 5.2: Exemplo de uma tabela de identificadores

Se o identificador é o nome de uma função, o seu endereço (`getTableAddress()`) é usado para criar instâncias dessa função aquando das suas invocações. O nodo que representa a invocação não tem representação na tabela de identificadores.

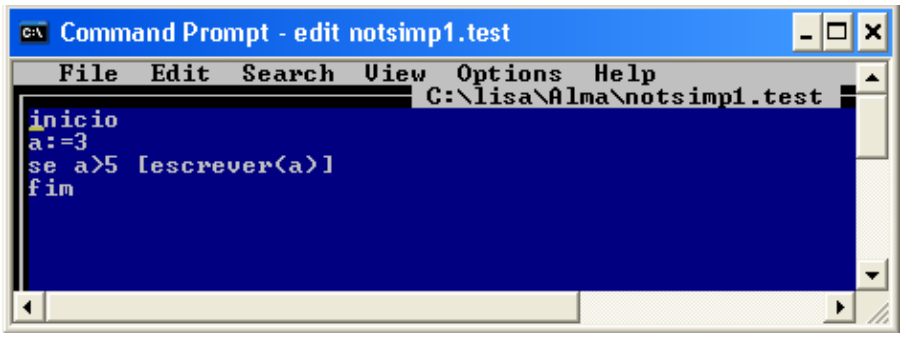

Figura 5.9: Texto fonte do primeiro exemplo

5.4.5 Visualizador do sistema LISA

Para mostrar os desenhos criados durante a aplicação das regras de visualização são usadas algumas classes do sistema LISA. Essas classes permitem a visualização desses desenhos num editor que resolve problemas de escalabilidade (com réguas vertical e horizontal) e produz também um desenho miniatura de toda a imagem que constitui a visualização e que dá a perceber a dimensão da mesma. Para demonstrar a capacidade deste visualizador, irão ser apresentados dois exemplos, escritos numa linguagem especificada pela gramática apresentada no apêndice B. Um outro exemplo de utilização é apresentado no apêndice E.

O primeiro exemplo corresponde à animação do programa apresentado na figura 5.9; algumas imagens da sua animação podem ser vistas nas figuras 5.10 e 5.11. Neste exemplo, o programa fonte contem uma atribuição e uma estrutura condicional. Um segundo exemplo, contem uma atribuição e uma estrutura repetitiva. O texto fonte está na figura 5.12 e parte da animação gerada pelo sistema Alma pode ser vista nas figuras 5.13,5.14,5.15 e 5.16.

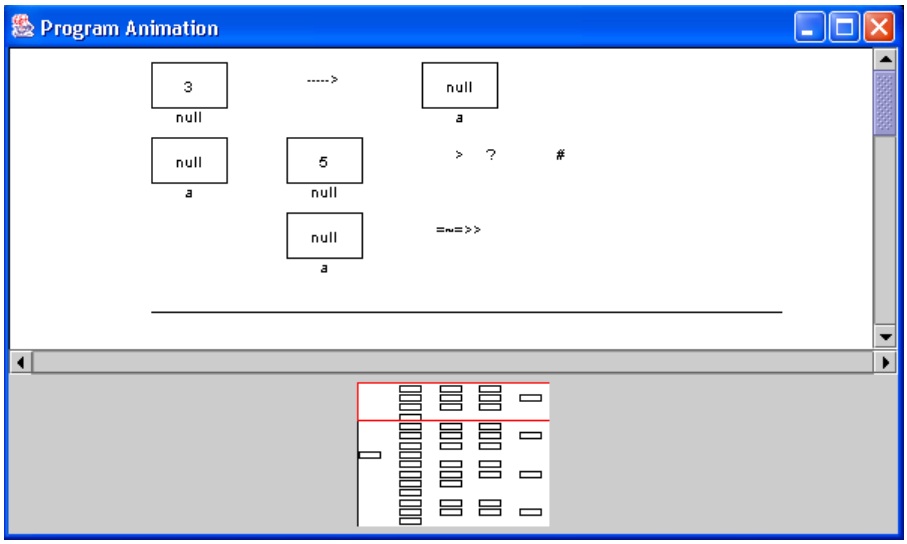

Figura 5.10: Imagem inicial da animação do primeiro exemplo

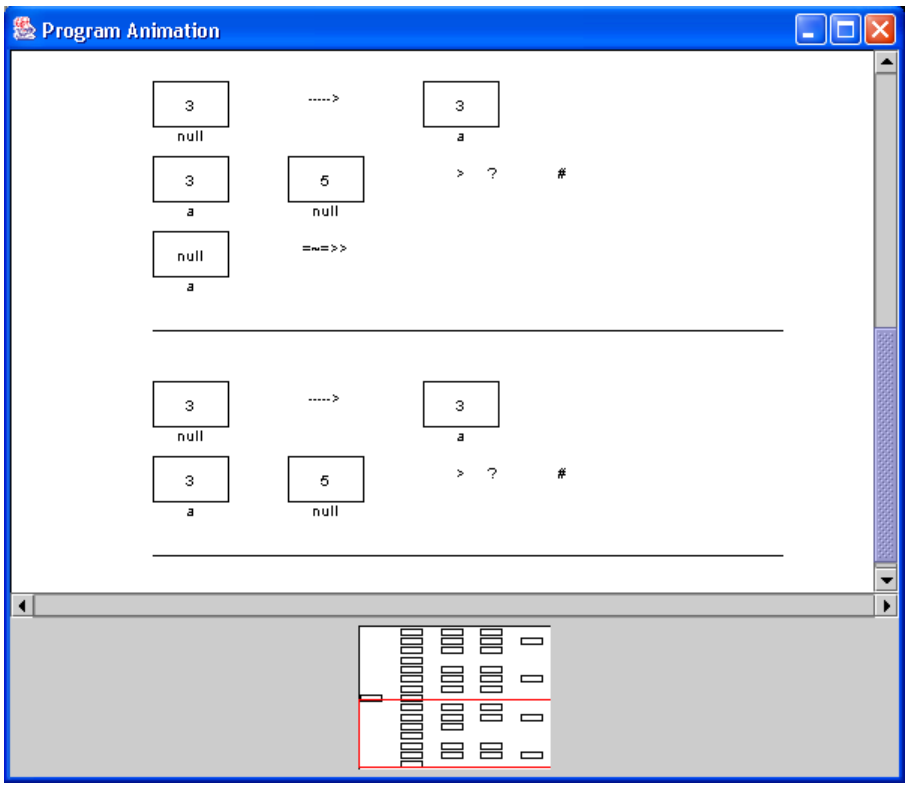

Figura 5.11: Últimas visualizações do primeiro exemplo

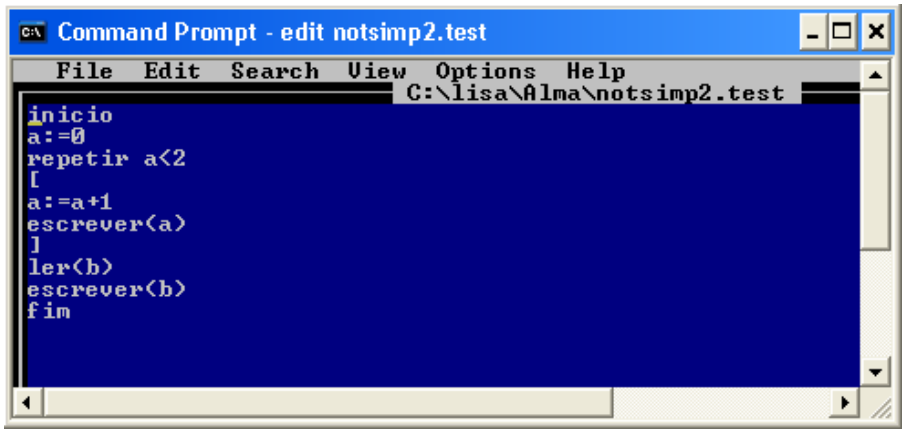

Figura 5.12: Texto fonte do segundo exemplo

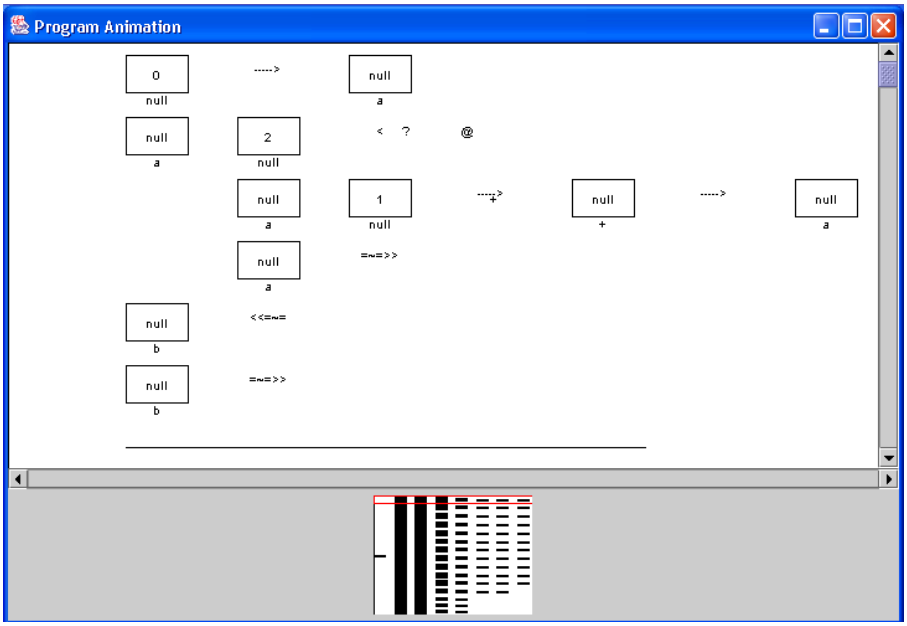

Figura 5.13: Primeira visualização do segundo exemplo

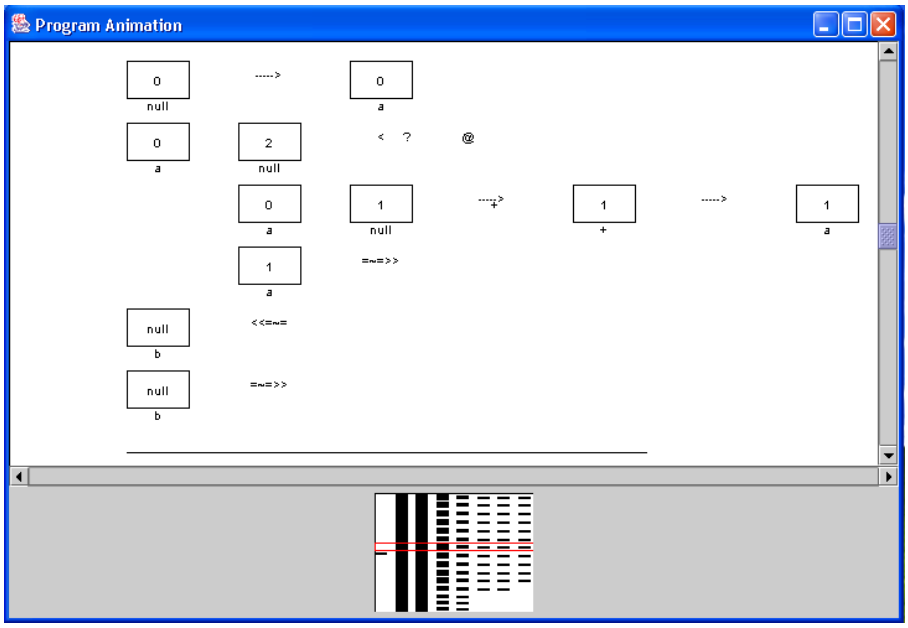

Figura 5.14: Visualização da primeira iteração do ciclo

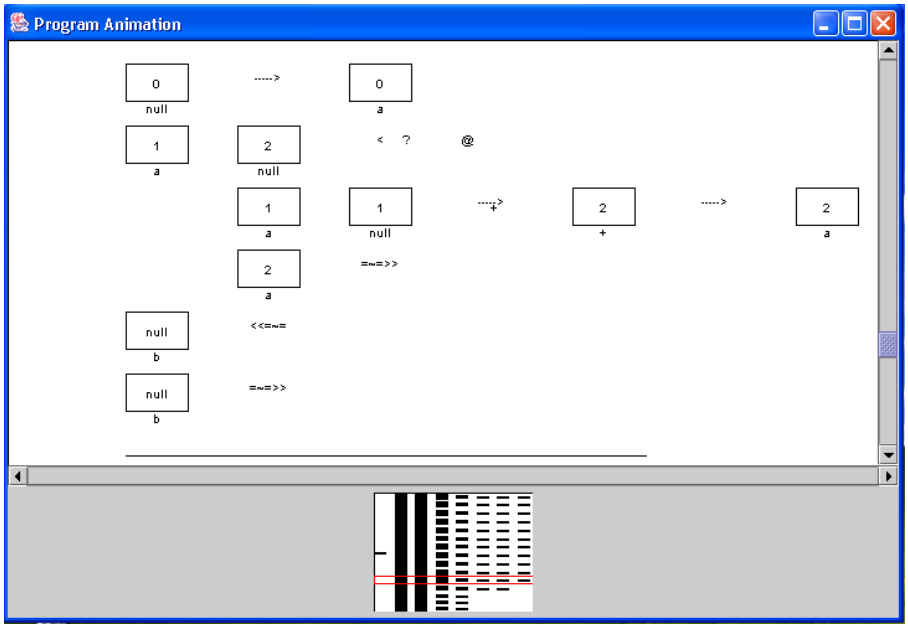

Figura 5.15: Visualização da segunda iteração do ciclo

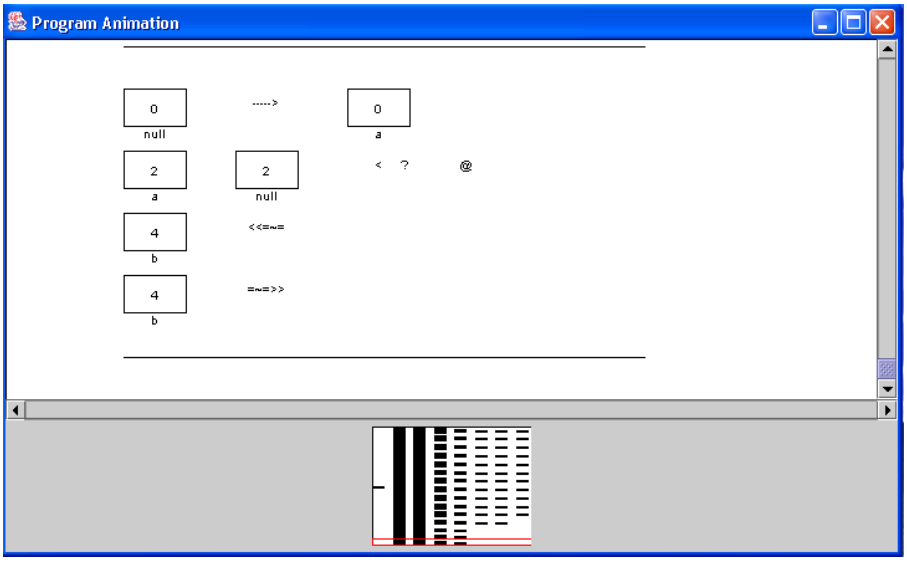

Figura 5.16: Última visualização da animação do segundo exemplo

5.5 O *back-end*: numa perspectiva de Máquinas Virtuais

Estando concluído um protótipo do sistema Alma é interessante ver o modelo do sistema tendo como base máquinas virtuais.

O processo de animação depende de duas tarefas distintas e independentes que, no entanto, devem estar sincronizadas: a reescrita e a visualização da DAST. Todo este processo trabalha sobre essa representação interna abstracta, usando diferentes tipos de especificações relacionadas com: o desenho a gerar; a reescrita da árvore; o controlo do *screen pointer*; o controlo do *tree pointer*; e, por último, o controlo global da animação gerada (ou seja, controlo do processo de sincronização entre a reescrita da árvore a a geração da visualização).

Durante a especificação do sistema Alma detectou-se a necessidade de definir quais os comandos de desenho que poderão ser utilizados na definição das regras de visualização (linguagem de desenho). Um outro conjunto de comandos era também preciso para especificar as alterações que a DAST sofre aquando da aplicação de uma regra de reescrita (linguagem de transformação de árvores).

Para além disso, sentiu-se a necessidade de criar um mecanismo de sincronização entre o processo de reescrita e o processo de visualização. Este mecanismo permite controlar o detalhe da animação, controlando o número de reescritas efectuadas antes de cada visualização. Para que este controlo possa ser feito facilmente, a sua implementação baseia-se em comandos (mais uma linguagem a definir, linguagem

de controlo de animação) que permitem calcular a frequência de amostragem.

Estas três linguagens *internas*, que tiveram de ser definidas, podem ser consideradas como casos de DSL (*Domain Specific Language*): foram criadas para sistematizar a programação das regras; para facilmente alterar os desenhos da animação; para especificar a semântica associada aos programas que queremos animar ou a frequência de amostragem da animação.

Na figura 5.17 estão esquematizadas as várias tarefas envolvidas na construção da animação. Cada linguagem de especificação ficará sujeita à respectiva máquina virtual, que interpreta o texto fonte e produz o respectivo resultado.

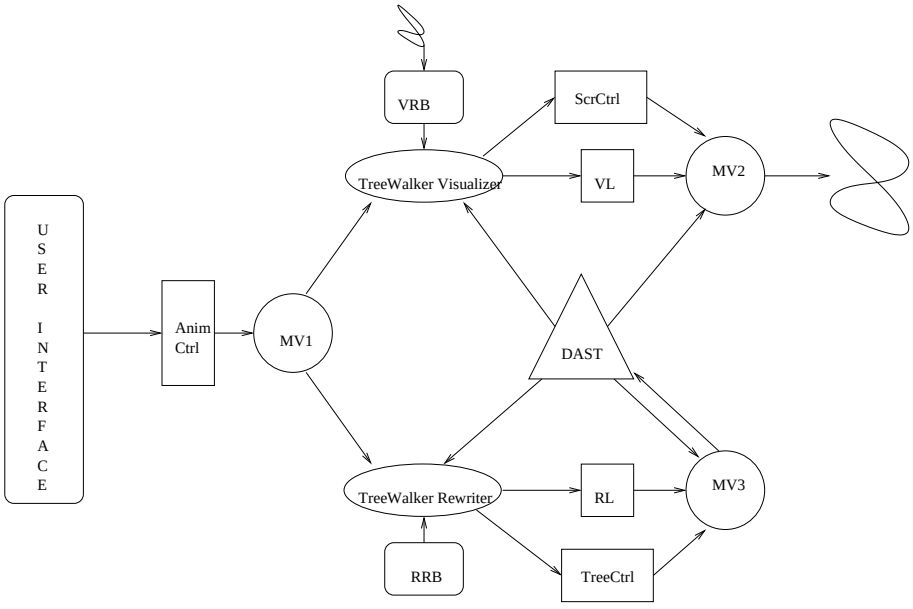

Figura 5.17: Processo de construção das visualizações

A linguagem de desenho (VL) contém as instruções responsáveis pelo desenho da árvore num dado instante. Esse desenho mostra um estado interno do programa fonte durante a sua execução.

```
VL           -> &
             | Shape VL
Shape        -> StructShp  |   OperationShp
StructShp    -> (conststhp|varshp|arrayshp|treeshp
             |queueshp|stackshp) tree_node
OperationShp-> (gen-opershp | relopershp) tree_node
             | readshp | writeshp
```

Estes comandos representam não só o desenho que irá ser produzido mas também os valores e etiquetas que terão que ser incluídos e que dependem dos atributos dos nodos da DAST (`tree_node`).
A linguagem de reescrita serve para especificar as alterações que a DAST terá que sofrer para representar um novo estado do programa fonte.

```
RL          -> &
            | RewStat RL
RewStat     -> CondStat | Stat
CondStat    -> Cond+ Stat+
Cond        -> Stat reloper Stat
Stat        -> setAtt tree_node att
            |  getAtt tree_node att
            |  copyNode tree_node tree_node
            |  delNode tree_node
            |  Stat oper Stat
```

Cada `RewStat` representa uma instrução de reescrita que pode estar condicionada ou não. Estes comandos representam todo o processo de reescrita: não só incluem comandos que efectuam a alteração de valores (alterações semânticas) como é o caso de `setAtt` como também as condições necessárias para que essas alterações sejam feitas. Os comandos `copyNode` e `delNode` implementam alterações sintácticas na DAST.
A linguagem de controlo do *screen pointer* serve para poder manipular as posições de desenho no ecrã, criar novas janelas; abrir e fechar janelas já existentes.

```
ScrCtrl     -> clear | new_wind coords int int int n_janela |
               close_wind int | set SP coords int |
               open_wind int | get SP int
```

A linguagem de controlo do *tree pointer* serve para controlar as travessias na árvore (usado, principalmente nas estruturas repetitivas); e para colocar marcas nos nodos que deverão ou não ser visualizados (para controlar o detalhe da visualização).

```
TreeCtrl    -> set TP tree_node | get TP |go_home |
               setvisible tree_node | setinvisible tree_node
```

A linguagem de controlo da animação serve para sincronizar o processo de reescrita com o processo de visualização que também pode servir para controlar o detalhe das animações.

```
AnimCtrl    -> rewrite tree_node |
               visualize tree_node
```

A especificação do sistema Alma já tem definido o posicionamento dos desenhos no ecrã. A disposição dos desenhos não será programável. Os desenhos aparecem sequencialmente tal como é sequencial a execução dos programas. O desenho de uma instrução básica (atribuições, leituras, escritas, invocação de funções, etc) ocupa uma linha. Se se tratar uma instrução condicional ou cíclica, o desenho ocupa uma linha para a condição seguida de um número linhas dependente do numero de instruções agrupadas. Sempre que há um agrupamento de instruções dentro de uma estrutura condicional ou cíclica é feita uma indentação em todas os desenhos relativos a essas instruções. Existem ainda alguns desenhos auxiliares que permitem distinguir uma instrução condicional da cíclica, assim como, para a distinção entre as instruções da primeira parte de uma instrução condicional (diz respeito ao `THEN`) e a segunda parte da mesma (diz respeito ao `ELSE`). Assim sendo, o *ScrCtrl* é feito com a ajuda do cálculo do nível de aninhamento (número de indentações) e do número de instruções agrupadas (número de linhas com indentação).

```
operrelShape : nome_oper,n_inst_agrupadas,nível_aninhamento -> desenho
```

O desenho associado a um operador relacional é definido na entidade `operrelShape` onde irá também ser armazenada informação sobre os desenhos das instruções agrupadas sob essa condição.
O chamado *TreeCtrl* é efectuado à custa de etiquetas postas nos nodos da DAST, de modo a que a travessia da árvore seja por vezes interrompida ou repetida nalguns nodos, dependendo do valor das condições das estruturas condicionais ou cíclicas.
Exemplos:

```
putName(``SKIP'') no nodo que encabeça o grupo de instruções
                   a ser ignorado
putName(``REP'') no nodo que encabeça o grupo de instruções
                 a ser atravessado novamente.
```

Estes comandos são usados nas regras de reescrita. Durante o processo de reescrita a árvore é atravessada seguindo uma política *pré-order*, mas quando é aplicada uma regra que detecta a existência de alguma destas etiquetas a travessia pode ser abortada ou repetida para alguns nodos.
Nesta nova perspectiva, ou visão, do *back-end*, podemos então dizer que o sistema Alma é construído à custa de cinco DSL's e inclui três máquinas virtuais com os seguintes objectivos:

- MV1 - para controlo do detalhe da animação; sincronização entre o processo de reescrita e o processo de visualização através da função `shownow()` do algoritmo de animação.

- MV2 - para interpretação dos comandos de desenho das regras de visualização.

- MV3 - para interpretação dos comandos de alteração dos atributos dos nodos que constam nas regras de reescrita.

Capítulo 6

Diferentes níveis de utilização do Sistema Alma

Como se disse em [VH03], após a implementação do protótipo do sistema Alma concluiu-se que, para abranger outros tipos de linguagens ou outros tipos de visualizações, bastaria acrescentar mais regras ao sistema. O sistema Alma tornou-se assim um sistema aberto, preparado para ser constantemente actualizado mediante as necessidades. Esta tarefa de actualização é bastante simples porque a implementação do sistema seguiu uma filosofia modular onde estão muito bem definidas as partes fixas e as que podem ser modificadas.

A ideia de criar um sistema aberto surge da necessidade de conjugar a generalidade do sistema com a sua adequação aos diversos tipos de problemas a resolver. Sendo assim, a parte fixa do sistema concede-lhe a generalidade pretendida e a outra parte, que poderá ou não ser modificada, concede-lhe a adequação necessária para obter bons resultados nas mais diversas situações.

Neste capítulo, irão ser apresentados três níveis de utilização do sistema. O tipo de utilização depende do tipo de utilizador: o utilizador final que usa o sistema para animar os seus mais diversos programas; o utilizador intermédio que pretende preparar o sistema para uma nova linguagem (sem fazer alterações ao sistema); e o utilizador que pretende acrescentar ao sistema novas potencialidades. Estes dois últimos implicam o acrescento de novas regras e conceitos, ou simplesmente a construção de um novo *front-end*.

6.1 Utilização básica do sistema Alma

O sistema Alma quando está preparado para receber programas escritos numa determinada linguagem pode ser aplicado a qualquer texto fonte dessa natureza e serão automaticamente abertas as janelas de animação. A partir do momento que o *front-end* foi construído, qualquer programa escrito nessa linguagem que entre no sistema Alma poderá ser animado, sendo gerado um conjunto de visualizações que representam o seu funcionamento. Para tal, basta escrever na linha de comando:

```
> java Compile nome_ficheiro.test
```

Sendo `nome_ficheiro.test`, o nome do ficheiro que contem o programa a ser animado.

6.2 Utilização avançada do sistema Alma

Nesta secção é apresentada um tabela que mostra a parte fixa do sistema Alma e a parte que pode ser alterada de modo a estender / modificar o comportamento do Alma.

Tabela de modificações/resultados

PARTE FIXA	PARTE EXTENSÍVEL	
	Pretende-se obter	Parte a alterar
Algoritmo de animação	$\neq$ **linguagens**	novo FE
	$\neq$ **nível de detalhe da animação**	função `shownow` do AA
Algoritmo de visualização	$\neq$ **tipos de visualizações**	regras de visualização
Algoritmo de reescrita	$\neq$ nível de detalhe das visual.	+
	novos desenhos	mecanismo de interpretação
Tabela de Identificadores	$\neq$ nível de abstracção	de desenhos
	$\neq$ **tipos de linguagens**(paradigmas)	nova semântica (RR)
Mecanismo de interpretação		+ novos nodos
das condições das regras		= novas regras de reescrita

Tabela 6.1: Modificações no sistema Alma

O sistema Alma é composto por partes fixas relacionadas com o mecanismo de aplicação de regras e geração da animação (algoritmos de animação, de visualização e de reescrita); outras partes podem ser incrementadas com novos *front-end*, novas regras, nova semântica ou novos nodos dependendo do pretendido. Se o utilizador do sistema pretende aplicá-lo a uma nova linguagem terá que escrever uma

extensão à gramática dessa linguagem, especificando o mapeamento entre os seus conceitos e os descritos pela gramática abstracta do Alma, para com ela gerar um novo *front-end*. Se o mapeamento for feito com sucesso, nada mais é necessário. O utilizador poderá submeter todos os programas que desejar desde que estes sigam a gramática da nova linguagem. No entanto, o utilizador poderá querer indicar o detalhe de animação que pretende, fazendo variar o período de amostragem ou indicando os nodos da DAST que tenciona ou não visualizar.

O tipo de visualização pode também ser controlado pelo utilizador à custa da criação de novas regras de visualização. Após ter sido criado o *front-end* para a sua linguagem, poderá querer associar novos desenhos aos nodos da DAST com o intuito de alterar o detalhe da visualização ou de usar novos ícones ou ainda (uma mistura dos dois anteriores) alterar o nível de abstracção. Para o utilizador que não conseguiu (não achou adequado) mapear todos os conceitos da sua linguagem nos nodos já conhecidos é também dada a possibilidade de criar novos nodos e novas regras de reescrita para lhes associar a parte semântica.

Seguindo os tópicos apresentados nesta tabela, nas secções seguintes, irão ser discutidos os vários tipos de intervenção no sistema Alma.

6.3 Construção de novos FE's

Para cada linguagem fonte terá que ser construído um FE respectivo para mapear os conceitos dessa nova linguagem nos símbolos da gramática abstracta do Alma. O utilizador deve definir a sua linguagem usando o sistema LISA. Para construir o *front-end*, o utilizador para além da gramática deve definir as acções semânticas que, neste caso, não traduzem o significado de cada frase. A parte semântica operacional da linguagem pode ser omitida e em vez disso, em cada produção são construídos nodos da linguagem abstracta do Alma que já têm a semântica bem definida.

Em princípio, convém recorrer ao sistema LISA para construir o *front-end* porque são usadas classes desse sistema para construir a DAST. Também será possível que outro sistema possa importar essas classes de modo a produzir o mesmo resultado final.

As funções, ou *templates* LISA, que poderão ser usadas para construir a DAST têm os seguintes nomes: `ALMA_ROOT`, `ALMA_STATS`, `ALMA_ASSIGN`, `ALMA_READ`, `ALMA_WRITE`, `ALMA_OPER`, `ALMA_RELOPER`, `ALMA_VAR`, `ALMA_CONST`, `ALMA_IDENT`.

Em conclusão, para que o sistema Alma esteja apto a ser usado para outras linguagens basta construir novo *front-end* que será responsável por traduzir todos os programas escritos nessa linguagem numa

DAST do Alma. Se houver algum conceito dessa nova linguagem que não possa ser expresso à custa dos conceitos já definidos no sistema Alma, terá que ser feita uma intervenção mais profunda ao sistema ao nível da geração de diferentes tipos de visualização, ou diferentes semânticas, como se vai explicar nas secções seguintes.

6.4 Diferentes tipos de animações/visualizações

A base de regras de visualização do Alma, poderá, a qualquer momento, ser enriquecida com novos desenhos para substituir as actuais (ou acrescentar). As novas regras pretendem cobrir algum tipo de linguagem fonte que, à partida, não se identifica com o paradigma imperativo ou semelhante. O protótipo que foi concluído gera animações de programas imperativos e outros que se possam mapear nos mesmos conceitos básicos, no entanto, é desejável que mesmo para este tipo de programas se possa gerar diferentes tipos de visualizações. Assim, regulando a frequência de amostragem e usando novos desenhos associados aos mesmos nodos, ou a nodos de mais alto nível, o sistema produzirá como resultado uma visualização diferente.

6.4.1 Diferente nível de detalhe da animação

O detalhe da animação depende do período de amostragem e da parte do programa que se pretende visualizar. O primeiro é uma questão de sincronização entre o processo de visualização e o processo de reescrita e é resolvido recorrendo a uma expressão booleana. O segundo está relacionado com a escolha dos nodos a visualizar.

Diferente frequência de amostragem

Numa visão mais detalhada a frequência de amostragem tem valor 1, ou seja, sempre que a árvore é reescrita é visualizada. No algoritmo principal de construção da animação, o processo de visualização é despoletado de acordo com o valor de uma função - `shownow()`. Esta função é booleana e permite controlar a frequência de amostragem devolvendo `V` ou `F`. Considerando um pequeno excerto de código:

```
...
int niter=0;
ss = visualize(n,ss);
n=rewrite(n);
```

```
niter++;
do{
   if (shownow(niter,x)) ss=visualize(n,ss);
   ...
   n=rewrite(n);
   niter++;
}while(...);
```

A variável `niter` representa o número de iterações. O período de amostragem, aqui representado pela variável `x`, indica o número (fixo) pretendido de reescritas para cada visualização. O resultado de fazer variar o `x` é uma animação composta por mais ou menos desenhos do programa; nenhuma informação é ocultada mas, de uma visualização para outra, ocorrerão `x` alterações. Se o `x` tiver uma valor elevado pode fazer com que a animação se torne incompreensível.
A função `shownow()` poderá ser definida da seguinte forma:

```
boolean shownow(int it, int x){
if(it%x==0) return true;
    else return false;
}
```

Este tipo de intervenção no sistema Alma pode ser feita pelo utilizador final, aquando da introdução do programa a animar, bastando para tal, indicar qual o valor de `x` pretendido. Esta facilidade deve ser incluída na interface final do sistema.
Para concluir, a frequência de amostragem faz variar globalmente a velocidade de animação, mas não faz distinção entre os diversos pontos do programa, ou seja, não permite variar a velocidade consoante o nível de interesse das visualizações geradas.
Poderá ser estudada a hipótese de definir valores de `x` para os vários tipos de instrução. A ideia é que o processo de reescrita possa alterar o valor de `x` consoante o tipo de instrução que está a reescrever. Neste caso, o utilizador final não escolherá a frequência de amostragem da animação, mas esta variará automaticamente consoante o nível de interesse das instruções.

Diferentes nodos a animar

O detalhe da animação depende do número de visualizações (frequência de amostragem) mas também da escolha das instruções, ou tipos de instruções, a visualizar. O utilizador final, dispondo de uma interface amigável, poderá indicar os nodos da DAST a visualizar, ou simplesmente indicar um tipo de instrução que não lhe interessa incluir na animação.
A nível interno do sistema Alma esta facilidade é simples de implementar, bastando colocar etiquetas `SKIP` (actualmente usada nas estruturas condicionais e cíclicas) nos nodos que não interessa visualizar. Este procedimento é efectuado antes de iniciar o processo de reescrita e visualização da DAST. Uma travessia inicial serve para etiquetar a árvore e prepará-la para ser usada.
Mais uma vez esta facilidade depende de uma interface visual expedita para o sistema Alma, que permita programar a animação.

6.4.2 Diferentes tipos de visualizações

Nesta secção irão ser apresentadas várias considerações sobre a geração de diferentes tipos de visualizações. O utilizador pode obter visualizações diferentes, associando novos desenhos às regras, conseguindo um maior ou menor nível de abstracção em função das associações nodos-desenhos que especificar. No entanto, e como já foi discutido em 6.3, a aplicação do sistema Alma a uma linguagem fonte muito diferente do habitual paradigma imperativo, implica a construção obrigatória de novas regras e, naturalmente, de novos desenhos.

Diferente nível de detalhe das visualizações

Pretende-se distinguir o nível de detalhe de uma visualização em relação ao detalhe de uma animação: considera-se que o detalhe de uma animação está directamente relacionado com o número de visualizações e o detalhe de uma visualização está relacionado com a quantidade de informação que se transmite com os desenhos produzidos.
Associando um conjunto de desenhos mais ou menos completo aos nodos representativos de um dado conceito básico, obteremos uma visualização mais ou menos detalhada desse mesmo conceito.
Este tipo de intervenção no sistema Alma implica a adição de novas regras de visualização onde ao mesmo conjunto de nodos, ficará associado um novo conjunto de primitivas de desenho.

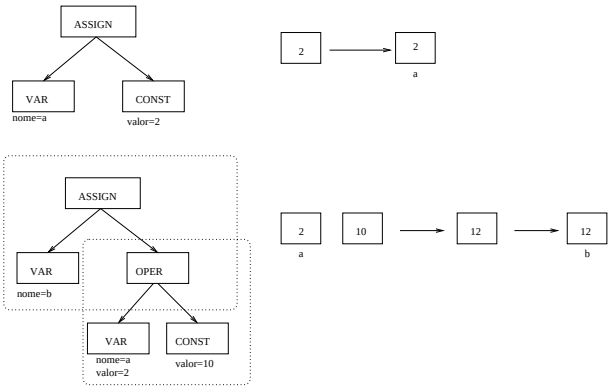

Figura 6.1: Última visualização gerada para o exemplo 6.4.1

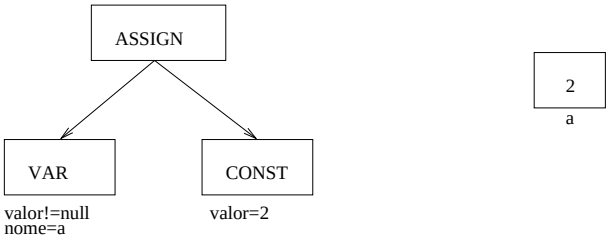

Figura 6.2: Uma visualização diferente para a primeira instrução do exemplo 6.4.1

A figura 6.1 mostra a visualização gerada para duas instruções de atribuição:

Exemplo 6.4.1 (Programa de entrada)

```
a=2
b=a+10
```

As figuras 6.2 e 6.3 representam as novas regras associadas às mesmas instruções. Estas modificações poderão ser feitas pelo projectista do sistema Alma ou pelo utilizador intermédio se lhe for disponibilizada uma interface para introdução de novas regras no sistema.

Novos desenhos

As primitivas de desenho pertencentes ao sistema Alma (por exemplo `varShape`, ou `operShape`) podem ser alteradas de modo a associar a esses conceitos básicos outros desenhos. Este tipo de modificação não implica qualquer mudança das regras de visualização porque os desenhos são gerados para os mesmos nodos, só a sua aparência é que muda.

Neste caso, este tipo de intervenção no sistema pode ser efectuada pelo projectista do sistema ou por

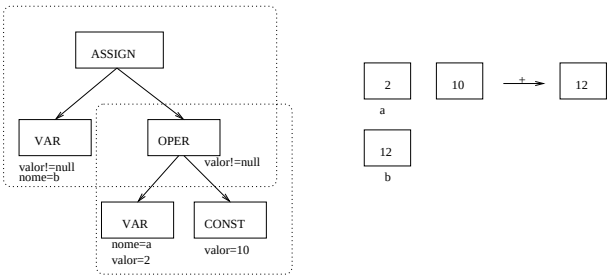

Figura 6.3: Uma visualização diferente para a segunda instrução do exemplo 6.4.1

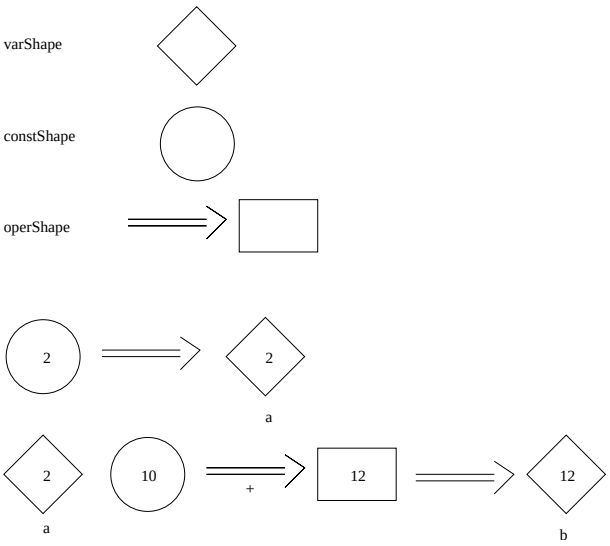

Figura 6.4: Nova visualização do exemplo 6.4.1 usando novas primitivas de desenho

um utilizador intermédio se a interface permitir a redefinição dessas primitivas. Um exemplo de uma intervenção deste tipo está representada na figura 6.4. Esta figura, comparada com a figura 6.1, apresenta novos desenhos para as primitivas `varShape`, `constShape` e `operShape` e o resultado de as associar aos nodos representativos do programa exemplo 6.4.1.

Diferente nível de abstracção

Criar uma animação com base em novos desenhos pode não ser assim tão fácil. Se se pretender usar desenhos mais abrangentes (que devem ser associados a nodos superiores), devem ser criadas novas regras de visualização. É necessário indicar quais os nodos abrangidos e os respectivos desenhos. Normalmente, um desenho mais abrangente torna-se mais abstracto, capaz de representar o conceito na sua

essência mas ignorando informação mais pormenorizada.

Pode-se afirmar que ao modificar os desenhos e o nível de detalhe das animações obtem-se um diferente nível de abstracção. Dependendo do objecto ou assunto que se pretende animar existe um nível de abstracção considerado o mais adequado. A ideia é então criar novas regras de visualização com o objectivo de associar desenhos mais abstractos a nodos pertencentes a níveis mais elevados da árvore de sintaxe.

No exemplo 6.4.2, existe um pequeno robot com a missão de limpar uma área rectangular e usa-se uma linguagem simples para descrever os seus movimentos. O robot pode mover-se para cima (`UP`), para baixo (`DOWN`), para a esquerda (`LEFT`) ou para a direita (`RIGHT`) um certo número de passos (1 passo = 1 unidade de comprimento).

Considera-se então o seguinte programa fonte onde a posição inicial - ponto com coordenadas (0,0) onde o robot é posto no começo da sua actividade - é o canto superior esquerdo do rectângulo.

Exemplo 6.4.2 (Programa do Robot)

```
xi = 0
yi = 0
DOWN 3
RIGHT 7
UP 2
LEFT 4
```

Para obter animações deste programa será necessário criar um *front-end* que mapeie os seus conceitos nos nodos da DAST. Na figura 6.5 está representada a DAST que foi gerada para o exemplo em causa e que, resulta do mapeamento das primeiras duas instruções no nodo `ASSIGN` e todas as outras no nodo `LST`. O nodo `LST` permite representar agrupamentos de elementos de tipos diferentes (neste caso, direcção do movimento com o número de passos).

Uma regra de reescrita calcula as coordenadas finais de cada movimento do robot e uma regra de visualização desenha o robot em cada uma das posições resultantes. Concretizando, a regra de reescrita é especificada da seguinte forma:

```
rew_rule(lstmov) = <a:lst, b:const, c:const>,
                   (getvalue(b)!=NULL),
                   <a:lst, b: const, c: const>,
                   { x=getTableVal(xi);
                     y=getTableVal(yi);
                     calculate(x,y,getValue(b),getValue(c))
```

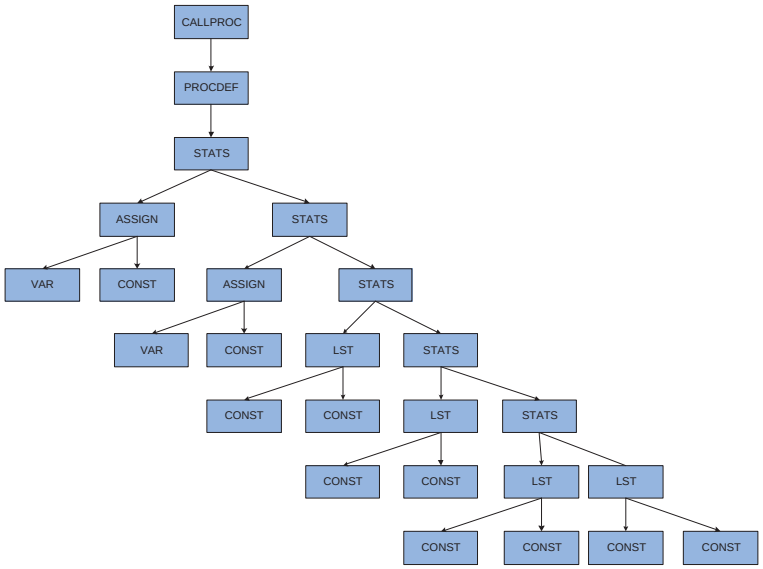

Figura 6.5: A DAST gerada para o exemplo do robot

Nome	Tipo	Classe	Valor	Endereço	TParam
xi	int	VAR	0	2	IN
yi	int	VAR	0	4	IN

Figura 6.6: A tabela de identificadores para o exemplo do robot

```
putTableVal(xi,x);
putTableVal(yi,y);}
```

A regra de reescrita, como já foi dito, tem como objectivo, para cada nodo do tipo `LST`, calcular os novos valores das coordenadas do robot e actualizar a tabela de identificadores (figura 6.6).

A regra de visualização consulta na tabela de identificadores os valores das coordenadas e desenha o robot na posição correcta. A função de desenho associada a esta regra pode até variar a imagem consoante o tipo de movimento do robot.

```
vis_rule(robot) = <a: procdef, b: stats>,
                  (),
                  <a:procdef, b:stats>,
                  {drawrobot(getValue(xi), getValue(yi))}
```

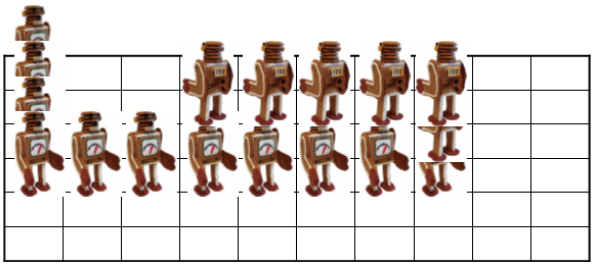

Figura 6.7: As várias imagens da animação do robot

O tipo de animação resultante pertence a um nível de abstracção mais elevado que as mostradas anteriormente. Na figura 6.7 surgem as imagens sobrepostas de toda a animação.

6.4.3 Diferente tipo de linguagem (paradigma)

Um tipo diferente de linguagem fonte (pertencente eventualmente a outro paradigma da programação) pode implicar definição de novas regras de visualização, não só com novos desenhos mas também com novas conjugações de nodos. Neste caso, o resultado do sistema será com certeza bastante diferente do habitual, apresentando níveis de abstracção, níveis de detalhe e tipos de desenhos adequados à linguagem fonte. Por outro lado, o utilizador pode, eventualmente, chegar à conclusão que nem todos os conceitos básicos da sua linguagem podem ser expressos com os nodos da gramática abstracta do Alma. Será então necessário definir novos nodos representativos desses conceitos. Neste caso, será fácil criar construtores para esses novos nodos não esquecendo de criar regras de reescrita que lhe concedam a semântica adequada.

Supondo que se pretende usar o sistema Alma para visualizar programas de um paradigma declarativo, lógico ou funcional, as regras de reescrita e de visualização devem ser repensadas. A visualização do código do programa pode ser feita à custa das regras de visualização existentes mas o novo cálculo semântico pode implicar novas regras de reescrita. No entanto, se se pretender, por exemplo, simular o funcionamento de um motor de prova de programas em Prolog, muitas das regras usadas para simular a invocação de subprogramas poderão ser aproveitadas. O resto desta subsecção será dedicado a explorar novos conceitos, nodos, regras e desenhos para visualizar programas em Prolog e animar o funcionamento de um motor de prova.

Primeiro exemplo de uma intervenção no sistema Alma para visualizar um paradigma diferente (lógico)

Para provar a generalidade da abordagem proposta, toma-se o caso dos programas em Prolog. Pretende-se apresentar exemplos de programas e suas visualizações, verificar quais são os novos conceitos envolvidos e explicar quais as novas regras a serem criadas.

Visualização de um programa em Prolog

Um programa em Prolog, tal como qualquer texto que represente um programa fonte em outra linguagem, é estático e, como tal, pode ser visualizado com apenas uma travessia da DAST. Considerando o conceito de variável, de constante, de predicado e de regra, consegue-se mostrar que as regras de visualização que já existem são suficientes para produzir esta animação.

Considerando que um predicado é uma função booleana, podemos representar este conceito usando o nodo `PROCDEF`. O nodo `PROCDEF` terá um atributo que corresponde ao nome do predicado e terá como descendentes uma lista de argumentos da cabeça e uma lista de átomos correspondentes ao corpo (se se tratar de uma regra). No caso de existir apenas uma lista de argumentos (facto) automaticamente o valor a ser posto no corpo do nodo `PROCDEF` é `true`. Quando se trata de uma regra e será necessário verificar a veracidade dos átomos do lado direito (associados a um nodo do tipo `STATS`) antes de concluir a veracidade do átomo do lado esquerdo. Assim, o nodo `PROCDEF` representa o conceito de predicado, quer seja regra, quer seja facto.

Numa primeira abordagem não será necessário criar novos nodos para o *back-end* do Alma. Assim, poderão ser usadas as mesmas regras de visualização já definidas no sistema. Construindo um *front-end* para uma linguagem Prolog, irão ser apresentadas a DAST e visualizações geradas para pequenos exemplos. O *front-end* é gerado associando às produções da gramática acções de construção de nodos da DAST.

O *front-end* da linguagem foi construído com base na especificação que se mostra no apêndice F, a qual traduz o mapeamentos apresentado na tabela 6.2.

MAPEAMENTO DE CONCEITOS/SÍMBOLOS	
Gramática Concreta	Gramática Abstracta
clausulas	STATS
facto	PROCDEF
regra	PROCDEF
argumentos	LST
atomos	STATS
atribuição	ASSIGN
operador	OPER
operador relacional	RELOPER
variável	VAR
constante	CONST

Tabela 6.2: Mapeamento entre conceitos da gramática concreta e símbolos da gramática abstracta do Alma

Tomando como exemplo um pequeno programa onde constam quatro factos e uma regra:

Exemplo 6.4.3 (Programa em Prolog)

```
mae(alda,joana).
mae(joana,joao).
pai(luis,pedro).
pai(luis,joana).
pais(M,P,E) :- mae(M,E), pai(P,E).
```

A árvore gerada após o reconhecimento feito pelo *front-end* especificado, está representada na figura 6.8. A visualização gerada pelo *back-end* é apresentada na figura 6.9.

Os predicados que são factos, ou lados esquerdos de regras, devem ser representados de forma diferente dos predicados do lado direito das regras. Estes últimos devem ser vistos como novas *queries* postas à base e serão assim mapeados em invocação de funções (`CALLPROC`) e não como uma definição. Em termos de regras de visualização não há diferença entre um nodo `CALLPROC` e um nodo `PROCDEF`. Iremos ver que na construção de uma animação, onde já são necessárias também as regras de reescrita, que a sua semântica é bastante diferente. A visualização de atribuições, expressões relacionais e aritméticas, é gerada à custa de regras que já existem e não precisam de ser alteradas. As regras de visualização asso-

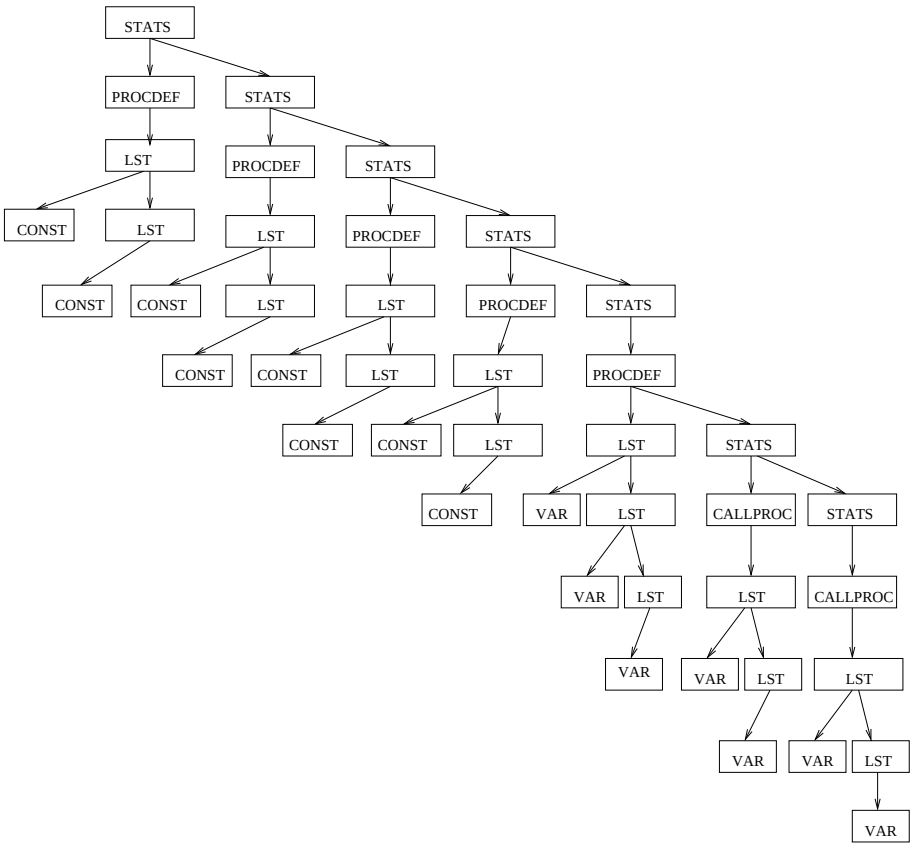

Figura 6.8: Árvore gerada pelo sistema LISA para o exemplo 6.4.3

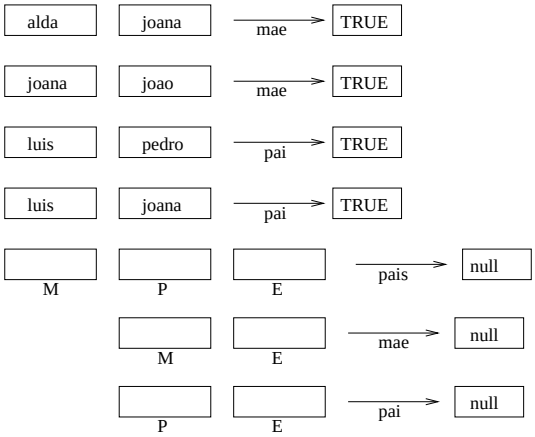

Figura 6.9: Visualização criada para o exemplo 6.4.3

ciadas aos nodos `PROCDEF` e `CALLPROC`, que foram aplicadas na criação da visualização apresentada, foram esquematizadas na figura 4.19.

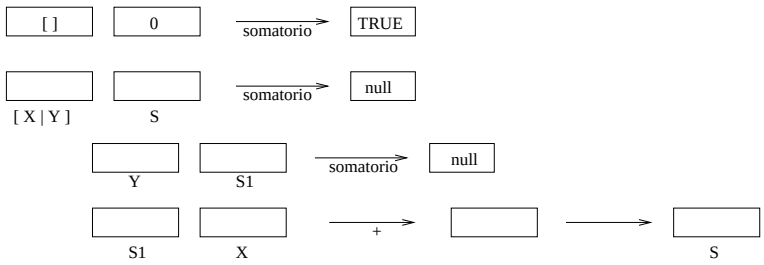

Figura 6.10: Visualização clássica para o exemplo 6.4.4

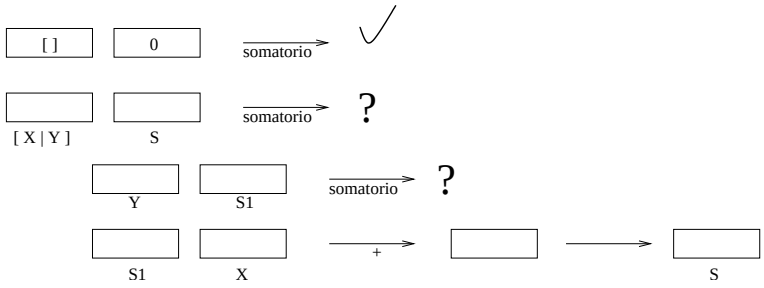

Figura 6.11: Nova visualização para o exemplo 6.4.4

Será apresentado ainda um segundo exemplo de programa em Prolog. Trata-se da definição de um predicado `somatorio`:

Exemplo 6.4.4 (Programa em Prolog)

```
somatorio([],0).
somatorio([X|Y],S) :- somatorio(Y,S1), S is S1 + X.
```

Não sendo necessário acrescentar mais regras nem fazer qualquer tipo de alteração, a visualização gerada para este exemplo é apresentada na figura 6.10.

No entanto, se se pretender tornar a visualização mais adequada ao paradigma lógico, as regras poderão especificar mapeamentos para desenhos diferentes. O mesmo exemplo poderia ser visualizado tal como é apresentado na figura 6.11.

Em conclusão, usando as regras existentes para o paradigma imperativo é possível visualizar programas em Prolog. Mas, é de notar que não conseguimos mais do que visualizar a base de conhecimento (BC), um conjunto de factos e regras que podem não ter qualquer relação entre eles. Por isso, parece ser muito mais interessante visualizar o processo de prova de um predicado.

Animação de um processo de prova

Dado um programa lógico, se se pretender uma animação do processo de prova em Prolog, criamos um conjunto de regras que retratam toda a dinâmica de aplicação de predicados e da unificação de átomos lógicos. Essas regras, que são aplicadas sobre os nodos já criados para a visualização do programa, vão especificar o cálculo de atributos necessário para simular a prova de uma *query* posta à base de factos e regras.

Na DAST estão representados factos e regras. Os factos são predicados cujo valor é sempre `verdadeiro` e são inseridos na base de conhecimento. Os predicados das regras não têm valor definido, vão tomando valores temporariamente com o objectivo de provar *queries*. A *query* no final terá o valor `falso` ou `verdadeiro` consoante aquilo que se conseguiu provar. Na tabela de Identificadores é então necessário armazenar os factos e os predicados do lado esquerdo das regras.

Considerando um predicado como sendo um subprograma (nodos `PROCDEF` e `CALLPROC`) cujo valor de retorno será `verdadeiro` ou `falso`, podemos adoptar soluções muito similares às usadas para resolver a animação dos subprogramas. Procede-se então à construção de uma árvore de execução a partir da árvore de programa (DAST), copiando as árvores dos predicados (enraízadas em nodos de definição) para expandir os nodos de invocação (*queries*). Assim, essa árvore de execução conterá apenas os predicados usados na prova, possibilitando a criação de visualizações do processo de prova e não da base de conhecimento.

Sendo cada predicado da BC mapeado num nodo `PROCDEF` e o predicado da *query* num nodo `CALLPROC`, as regras de reescrita serão muito semelhantes às usadas para calcular a invocação de subprogramas. No entanto, a escolha do predicado do qual se deve criar uma instância para provar a *query* deve ter em conta a possibilidade de haver mais do que uma hipótese. O processo de unificação também sofrerá algumas alterações na medida em que se pretende a unificação de argumentos da *query* com os argumentos das regras.

Considerando a BC do exemplo 6.4.3, já apresentado, pretende-se animar a prova do seguinte predicado:

Exemplo 6.4.5 (Query posta à base)

```
?- pais(M,P,joana).
```

Supondo que a Tabela de Identificadores foi preenchida com informação sobre factos e regras da base de conhecimento aquando da construção da DAST, a árvore de execução, apresentada na figura 6.12, descreve a regra usada para provar o predicado. O lado direito da regra implica a prova de mais dois predicados.

A árvore de execução irá ser novamente alterada (figura 6.13) quando substituirmos os nodos `CALLPROC`

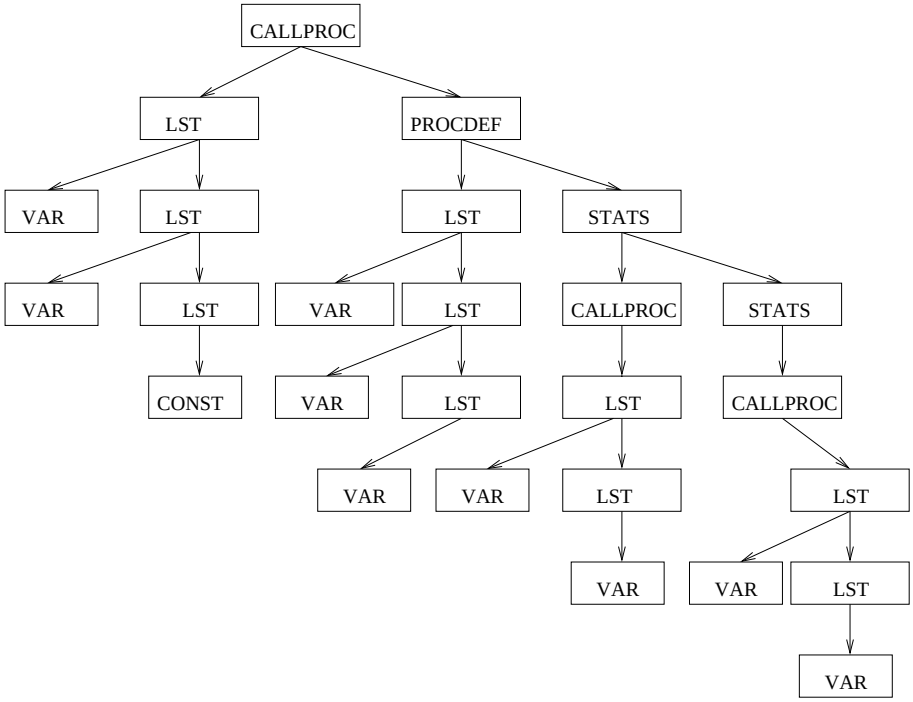

Figura 6.12: Árvore de execução para o exemplo 6.4.5

por nodos `PROCDEF`.

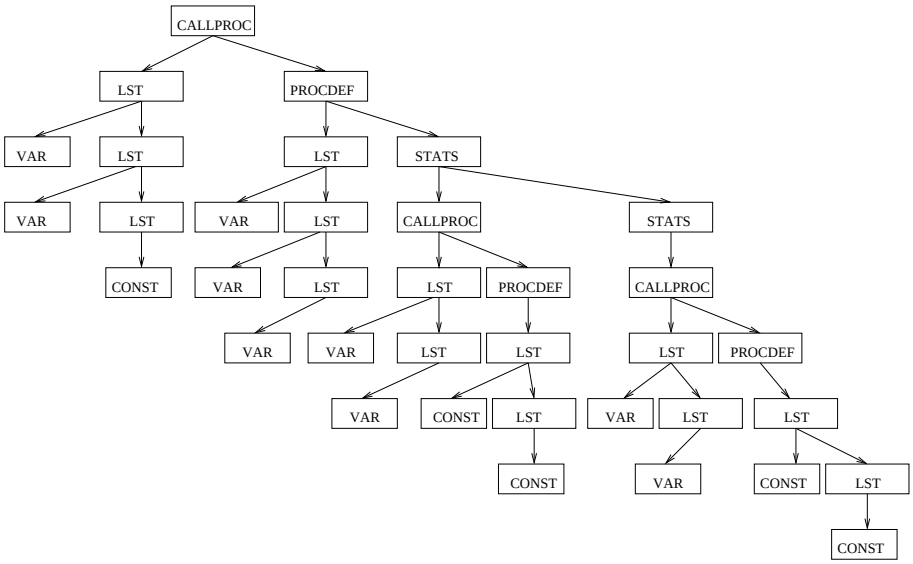

Figura 6.13: Nova árvore de execução para o exemplo 6.4.5

A animação gerada pelo sistema ALMA a partir desta árvore está esquematizada na figura 6.14. A

visualização não representa toda a DAST, mas apenas os predicados da *query* e os predicados necessários para a provar. Para que esta visualização seja gerada é necessário criar algumas novas regras de reescrita.

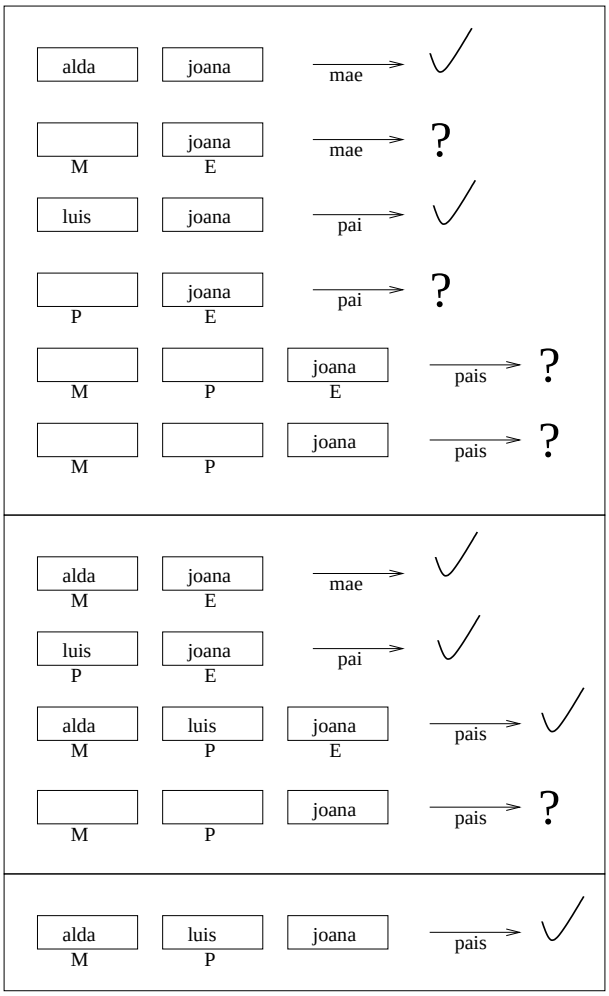

Figura 6.14: Animação criada pelo sistema ALMA

Os nodos `PROCDEF`, que representam factos, têm o valor verdadeiro e todos os outros valores de predicados são calculados em função destes. Quando todo o lado direito de uma regra fica provado, o lado esquerdo toma o valor `verdadeiro`. A escolha da regra a utilizar consiste em percorrer todas as regras da base de conhecimento e tentar fazer o *match* da *query* com o lado esquerdo de alguma. Ao copiar os nodos do `PROCDEF` adequado é criada uma tabela local a essa invocação que, mais tarde, irá ser unificada com os argumentos do predicado a provar.

Considera-se então vários grupos de regras: umas para propagação de valores e unificação de predicados (ver figura 6.15); outras para cálculo de valores (usadas já no paradigma imperativo); um outro conjunto

de regras para construir a árvore de execução (escolher para cada `CALLPROC` o respectivo `PROCDEF`). Cada `PROCDEF` pode conter, por sua vez, nodos `CALLPROC` que também serão associados aos respectivos `PROCDEF`. Essas associações baseiam-se na Tabela de Identificadores e quando estiverem feitas a árvore de execução está completa. As regras para construção da árvore de execução são as mesmas que foram usadas para a animação de subprogramas e foram apresentadas na figura 4.15. No entanto, terá, neste caso, que ser guardado em cada `CALLPROC` uma *stack* de endereços de nodos `PROCDEF` que poderão ser aplicados; os elementos irão sendo retirados da *stack* numa tentativa de atribuir o valor `verdadeiro` ao predicado; quando é necessário provar vários predicados (lado direito de uma regra) poderá acontecer que a impossibilidade de provar um predicado seguinte implique voltar ao predicado anterior (que já tinha o valor verdadeiro) e procurar outra solução.

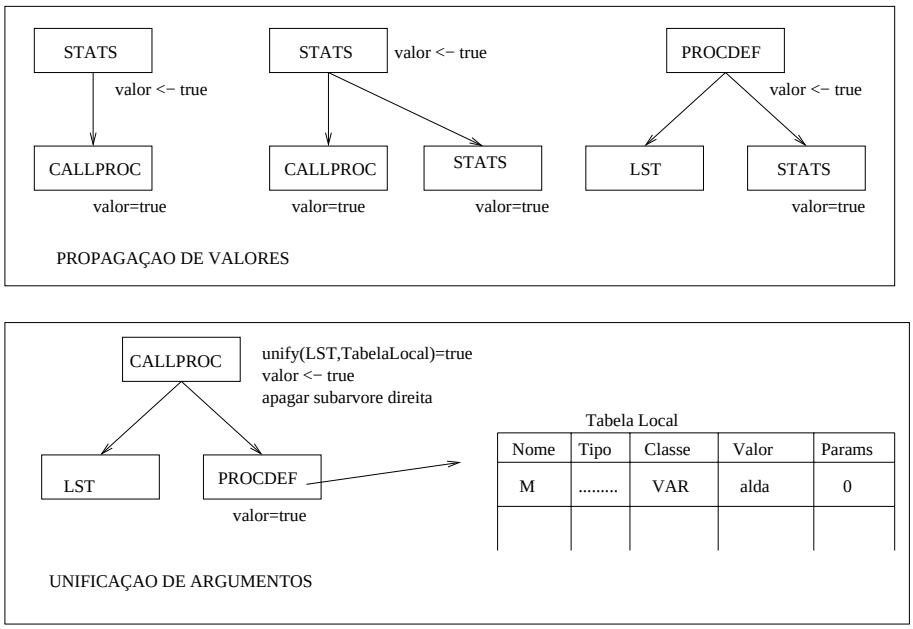

Figura 6.15: Regras de reescrita para unificação de predicados e propagação de valores

Há alguns predicados que merecem atenção especial: é o caso do `repeat` e do `!`(cut). O predicado `repeat` pode ser mapeado no nodo `WHILE` da gramática abstracta do Alma. O cut (!) implementa uma espécie de estrutura condicional, podendo ser mapeado com algumas restrições no nodo `IF` da gramática do Alma.

Por outro lado, um predicado *recursivo* pode ser representado usando os nodos abstractos já existentes e a semântica associada às suas regras de reescrita é igual à usada para outros predicados.

Segundo exemplo de uma intervenção no sistema Alma para visualizar um paradigma diferente (funcional)

Neste exemplo mostra-se mais uma vez a generalidade do sistema na construção de animações de linguagens pertencentes a outros paradigmas.

Visualização de um programa em Haskell

Um programa em Haskell, tal como nos outros paradigmas, é estático e é a invocação que lhe confere a dinâmica que se pretende animar. O exemplo 6.4.6, apresentado abaixo, calcula o tamanho de uma lista de elementos e pretende-se obter a animação desse cálculo aplicado a um caso concreto.

Exemplo 6.4.6 (Programa em Haskell)

```
len::[a]_Integer
len[] = 0
len(x:xs) = 1 + len xs

>len[2,3,5]
```

Considerando que a função invocada está definida no programa fonte, o nodo `CALLPROC`, que a representa, irá ser substituído por um nodo `PROCDEF` cujos nodos descendentes representam a definição da função. Assim, um programa em haskell, será visto como um conjunto de funções que serão chamadas sempre que ocorre uma sua invocação.

As regras de reescrita associadas a este tipo de nodos são as mesmas que são usadas na invocação de subprogramas no paradigma imperativo.

A DASTgerada para o exemplo 6.4.6 é mostrada na figura 6.16. Os nodos `LST` representam conjuntos de parâmetros quer na invocação das funções quer na sua definição.

As imagens da animação vão mostrando a execução das várias funções até chegar ao resultado final. A última imagem da animação será semelhante à apresentada na figura 6.17.

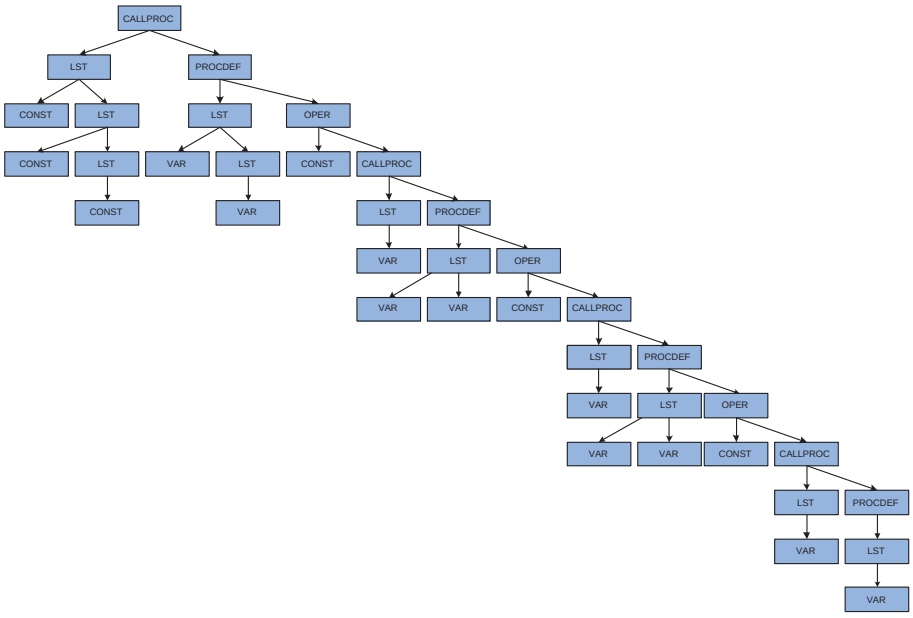

Figura 6.16: A DAST gerada para o programa em Haskell

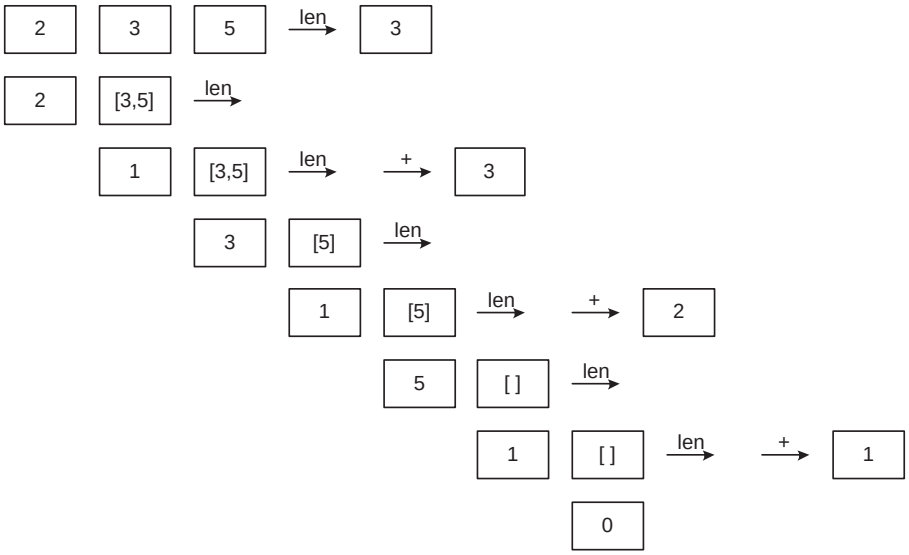

Figura 6.17: Resultado final da animação do programa em haskell

6.4.4 Como programar o que se quer acrescentar?

Para acrescentar ao sistema Alma novas produções da gramática abstracta (o que corresponde à criação de novos nodos da DAST); novas regras de reescrita ou de visualização; ou ainda novos desenhos, é necessário seguir alguns esquemas já definidos. Sendo o sistema Alma implementado usando a tec-

nologia Java, os ficheiros (Java) que o constituem definem cada uma das partes da sua arquitectura: os templates para criação de nodos; as regras de reescrita e de visualização; as máquinas virtuais de interpretação das regras (condições e acções); e, por último, as primitivas de desenho. Para cada caso, irá ser apresentado o *template* adequado. As partes do *template* que podem ser alteradas estão escritas em maiúsculas e entre #'s.

Novos construtores de nodos

No caso de ser um nodo folha, o *template* a usar será:

```
package  Alma;
import  Lisa.Parser.*;

public  class  #NOME DO CONSTRUCTOR#  extends  CAlmaNode  {
    public  #NOME DO CONSTRUCTOR# (#atributos NOME,TIPO e VALOR#)  {

        super(new CParseSymbolTerm(#NOME DO SIMBOLO#,22),
                    #NOME#,  #VALOR#,  #TIPO#);
        setProdNumber(#NUMERO DA PRODUÇÃO#);
    }
}
```

A função para ser usada na construção do *front-end* evita que o utilizador saiba o nome dos atributos usados, só tem que aplicar a função aos símbolos da sua gramática. A função define-se da seguinte forma:

```
template  <attributes  X_in,  Y_in, Z_in, k_in>
compute  ALMA_#NOME DO NODO#
{
    X_in.tree = new  Alma.#NOME DO CONSTRUTOR#(Y_in, Z_in, k_in);
}
```

No caso de ser um nodo intermédio, o *template* a usar será:

```
package  Alma;
import  Lisa.Parser.*;

public  class  #NOME DO CONSTRUTOR#  extends  CAlmaNode  {
```

```
    public  #NOME DO CONSTRUTOR#(#NODOS FILHOS#)  {

        super(new CParseSymbolNonTerm(#NOME DO SIMBOLO#,18),
                    ''null'',''null'',''null'');
        setProdNumber(#NUMERO DA PRODUÇÃO#);
        addNode(#NODOS FILHOS#);
    }
}

template <attributes X_in, Y_in,  Z_in, k_in>
compute ALMA_#NOME DO NODO#
{
    X_in.tree = new  Alma.#NOME DO CONSTRUTOR#(Y_in.tree,
                                    Z_in.tree,k_in.tree);
}
```

Os símbolos intermédios terão normalmente os atributos `NOME`, `VALOR` e `TIPO` inicializados a `''null''`. É de notar que cada nodo pode ter um, dois ou, no máximo três filhos. Estes templates referem-se sempre ao máximo permitido.

Novas regras de visualização

O código Java a usar para criação de uma regra de visualização deve indicar quais os nodos e quais os desenhos associados. Para ilustrar o processo a seguir quando se quer acrescentar uma nova regra, mostra-se abaixo um exemplo correspondente à regra de visualização de uma atribuição. Aqui são especificados os nodos `ASSIGN`, `VAR` e `EXP`, os seus atributos e respectivos desenhos.

```
/* NODO PAI */
       CParseSymbol  s11=  new  CParseSymbolNonTerm("ASSIGN",7);
       CRuleNode  r11 =  new  CRuleNode(s11,null,18);

/* NODO FILHO */
       CParseSymbol  s12=  new  CParseSymbolTerm("VAR",4,false);
       Object[]  ats11=new  Object[2];
       for(int  r=0;r<2;r++)  ats11[r]=null;
       expression  epr11=  new  expression("reading","name",aux0);
       ats11[0]=epr11;
```

```
        expression epr12= new expression("reading","value",aux0);
        ats11[1]=epr12;
        CRuleNode r12 = new CRuleNode(s12,ats11,22);

/* NODO FILHO */
        CParseSymbol s13= new CParseSymbolNonTerm("EXP",30);
        Object[] ats12=new Object[3];
        for(int r=0;r<3;r++) ats12[r]=null;
        expression ep11= new expression("==","type",aux1);
        ats12[0]=ep11;
        expression epr13 = new expression("reading","name",aux0);
        ats12[1]=epr13;
        expression epr14 = new expression("reading","value",aux0);
        ats12[2]=epr14;
        CRuleNode r13 = new CRuleNode(s13,ats12,30);

 /* LIGAÇÃO ENTRE NODOS */
        CRuleNode[] rs1= new CRuleNode[2];
        rs1[0]=r12;
        rs1[1]=r13;
        r11.setNodes(rs1);

 /* DESENHOS */
        draw w11= new draw("varShape",2,3);
        draw w12 = new draw("assignShape");
        draw w13 = new draw("varShape",0,1);
        drawing dr1= new drawing(3);
        dr1.putin(w11);
        dr1.putin(w12);
        dr1.putin(w13);

 /* INSERIR REGRA NA BASE DE REGRAS */
        insertRule(r11,dr1);
```

Na última parte da regra são especificados os desenhos e os índices onde encontra a informação sobre as variáveis, constantes e operações.

Novas regras de reescrita

A regra de reescrita de uma atribuição usa exactamente os mesmos nodos mas as acções são especificadas no próprio nodo.

```
/* NODO PAI */
      CParseSymbol  s11=  new  CParseSymbolNonTerm("ASSIGN",7);
      CRuleNode  r11 =  new  CRuleNode(s11,null,18);

/* NODO FILHO */
      CParseSymbol  s12=  new  CParseSymbolTerm("VAR",4,false);
      Object[]  ats11=new  Object[3];
      for(int  r=0;r<3;r++)  ats11[r]=null;
      expression  epr11=  new  expression("==","value",aux0);
      ats11[0]=epr11;
      action  epr12=  new  action("writing","value");
      ats11[1]=epr12;
      action epr13= new action("puttable","value");
      ats11[2]=epr13;
      CRuleNode  r12 =  new  CRuleNode(s12,ats11,22);

/* NODO FILHO */
      CParseSymbol  s13=  new  CParseSymbolNonTerm("EXP",30);
      Object[]  ats12=new  Object[3];
      for(int  r=0;r<3;r++)  ats12[r]=null;
      expression  epr14=  new  expression("==","type",aux1);
      ats12[0]=epr14;
      expression  epr15=  new  expression("!=","value",aux0);
      ats12[1]=epr15;
      expression  epr16=  new  expression("reading","value",aux0);
      ats12[2]=epr16;
      CRuleNode  r13 =  new  CRuleNode(s13,ats12,30);

/* LIGAÇÃO ENTRE NODOS */
      CRuleNode[]  rs1=  new  CRuleNode[2];
      rs1[0]=r12;
      rs1[1]=r13;
```

```
        r11.setNodes(rs1);

  /* INSERIR REGRA NA BASE DE REGRAS */
        insertRule(r11);
```

Pode acontecer termos que usar operações novas: sendo assim, é necessário acrescentá-las no método `changing` da classe das regras de reescrita:

```
  public CTreeNode changing(CTreeNode n,Object[] exps){
   ....................................

   for(;((i<exps.length) && (exps[i] instanceof action));i++){

   if ((((action)exps[i]).op).equals("writing")){

      Object[] params=new Object[2];
      if (global[1]==null){
         /* atribuição de uma variável para outra variável */
         ((Alma.CAlmaNode)n).setValue(global[0]);
      }
      else {params[0]=global[1];
            params[1]=global[2];
            /* atribuição do resultado de uma operação a uma variável */


      /* OPERAÇÕES ARITMÉTICAS, RELACIONAIS e LÓGICAS */
            if (global[0].equals("+"))
               ((Alma.CAlmaNode)n).setValue(String.valueOf(
                                    (Integer.parseInt(global[1])) +
                                    (Integer.parseInt(global[2]))));
            else if (global[0].equals("-"))
                 ..........................................................

            else if (global[0].equals(">"))
                    ((Alma.CAlmaNode)n).setValue(String.valueOf(
                                         (Integer.parseInt(global[1])) >
```

```
                                    (Integer.parseInt(global[2]))));
        else if (global[0].equals("<"))
                 ..........................................................

        else ((Alma.CAlmaNode)n).setValue((String)global[1] +
             (String)global[0] + (String)global[2]);

             }

    }
```

A máquina virtual que interpreta os comandos de reescrita encontra-se na mesma classe mas é fixa e está implementada com o seguinte código:

```
        ..............................................
        else if ((((action)exps[i]).op).equals("toread")){
                String val="";
                DataInputStream ent= new DataInputStream(System.in);
                System.out.print("Introduza o valor do parametro:");
                try{
                   val=ent.readLine();
                }catch (IOException e){
                System.out.println("Nao leu");}
                ((Alma.CAlmaNode)n).setValue(val);
        }
        else if ((((action)exps[i]).op).equals("towrite")){
                System.out.println(((Alma.CAlmaNode)n).getValue());
        }
        else if ((((action)exps[i]).op).equals("puttable")){
                ats[1]=((Alma.CAlmaNode)n).getValue();
                (Backend.tab).putTable((String)ats[0],(Object)ats[1]);
        }
        else if ((((action)exps[i]).op).equals("putname")){
                ((Alma.CAlmaNode)n).setName(((action)exps[i]).arg);
        }
        else if ((((action)exps[i]).op).equals("putnode")){
                ((Alma.CAlmaNode)n).setValue(String.valueOf(
```

```
                    (Backend.tab).getTable(ats[0])));
            }

    }
```

Novos desenhos

Cada conceito representado na DAST terá um desenho associado. A definição de desenhos de novos nodos ou a definição de novos desenhos para nodos existentes, é feita usando o seguinte *Template*:

```
import  java.awt.*;
public  class  #NOME DO DESENHO#  extends  Shape{
public  #NOME DO DESENHO#(#atributos NOME e VALOR caso tenha#){
    super(#atributos NOME e VALOR caso tenha#);
}
public void  draw(Graphics  g,FontMetrics metrics,int offx,int offy,
                  int posx,int posy,int width,int height){
       g.drawString(NOME...........);
       g.drawRect(.............);
       .........
    }
}
```

Cada desenho representa um elemento do programa que pode ser, por exemplo, uma variável, uma operação ou uma função. Cada um desses elementos tem associado um método `draw()` que permite definir a sua imagem na visualização. Ao criarmos uma nova função de desenho, temos que fazer com que a máquina virtual de interpretação de desenhos também a reconheça. De seguida é apresentado um excerto desse código que se encontra na classe das regras de visualização.

```
public  Vector  draw(Vector ss){
   ..............................
   for(int  p=0;  p<actions.num;  p++){
      ..............................
      if(actions.stats[p].name.equals("varShape")){
           if  (actions.stats[p].par1!=-1){
                a=global[actions.stats[p].par1];
           }
```

```
            if  (actions.stats[p].par2!=-1){
                 b=global[actions.stats[p].par2];
            }
            ss.addElement(new  varShape(a,(Object)b));
        }
        else  if (actions.stats[p].name.equals("operRelShape")){
              a=global[actions.stats[p].par1];
              ss.addElement(new operrelShape(a,conditions[1],conditions[2]));
              if ((conditions[0]).equals("W"))
                     ss.addElement(new circShape());
              else if((conditions[0]).equals("F"))
                     ss.addElement(new condShape());
              else if((conditions[0]).equals("I"))
                     ss.addElement(new condShape());
        }
        else if (actions.stats[p].name.equals("readShape"))
                     ss.addElement(new  readShape());
        else if (actions.stats[p].name.equals("assignShape"))
                     ss.addElement(new  assignShape());
        else  if (actions.stats[p].name.equals("operShape")){
                  if  (actions.stats[p].par1!=-1){
                      a=global[actions.stats[p].par1];
                  }
                  ss.addElement(new  operShape(a));
        }
        else  if  (actions.stats[p].name.equals("writeShape"))
                  ss.addElement(new  writeShape());

    }
         clean(global);
         return ss;
    }

}
```

Capítulo 7

Conclusão

Este capítulo pretende encerrar esta dissertação sobre o desenho e desenvolvimento de um sistema genérico de animação de programas. Esse projecto consistiu no estudo de viabilidade dessa construção e na concepção, especificação, implementação e manutenção desse mesmo sistema.

No final da escrita desta dissertação, tomou-se conhecimento de um sistema de visualização de programas criado em 1999 com algumas semelhanças com o sistema Alma. Em [Gra99] pretende-se provar que Prolog é uma boa linguagem de especificação de sistemas de visualização de software. O sistema construído chama-se `Vmax` e baseia-se em *mappings* formais especificados em Prolog para analisar o código fonte (em Java), extrair informação de execução e criar uma base de dados do programa. Para gerar as vistas são postas *queries* a essa base de dados de modo a conseguir determinadas informações sobre o programa. Os *mappings* especificados permitem obter a informação a visualizar, qual o conteúdo visual associado e quais as restrições gráficas dos diversos componentes. Combinando de outra forma os diferentes *mappings* obtém-se visualizações distintas. O autor afirma que seria possível a visualização de programas escritos noutras linguagens através da construção de novos *front-end* para Prolog.É possível então estabelecer algum paralelismo entre este sistema e o Alma, na medida em que usa uma linguagem intermédia e um modelo formal baseado em *mappings* para construir as visualizações. Este paralelismo vem reforçar as potencialidades da abordagem usada na construção do sistema proposto nesta tese.

7.1 Objectivos atingidos

Os objectivos principais deste estudo prendem-se com a necessidade de provar a tese que foi proposta:*É possível gerar automaticamente animações de programas independentemente da linguagem fonte*. Neste

âmbito foi feito:

- um estudo da área em causa, procurando analisar o actual estado da arte; neste contexto, e para sistematizar o estudo, foi criado um sistema de classificação das aplicações existentes agrupando-as por tipos; daqui resultou a identificação de uma lacuna a determinado nível.
- concepção de um novo sistema de animação (Alma) apresentando pormenores da sua arquitectura;
- especificação do sistema Alma: definição da linguagem abstracta, usada para representação interna (intermédia) dos programas fonte e sua semântica operacional; definição dos *templates* para as regras de visualização e de reescrita; criação dos algoritmos de visualização da árvore e de geração das animações; definição de tabela de identificadores.
- desenvolvimento do sistema Alma: implementação em Java do *back-end* do sistema de acordo com as especificações (criação de um protótipo).
- criação de dois *front-end* para testar a capacidade de criação das animações para duas linguagens diferentes;
- apresentação dos diferentes níveis de utilização do sistema Alma, na medida em que é completamente transparente para o utilizador final mas será um sistema aberto para um utilizador intermédio que pretende adaptá-lo às suas necessidades mais específicas.

7.2 Objectivos a alcançar em tempos futuros

Visto estar implementado apenas um protótipo do sistema, uma das tarefas futuras e mais urgentes será finalizar o seu desenvolvimento criando interfaces amigáveis para que o utilizador intermédio possa adicionar novas regras, novas primitivas de desenho ou novos nodos sem qualquer dificuldade. Assim, é necessário completar a janela de programação da animação para que seja possível:

- controlar o nível de detalhe da animação
- usar desenhos da DAST para escolha de nodos (partes do programa) a visualizar
- introduzir novas regras de visualização e de reescrita nas bases de regras do sistema
  - criar novas funções de desenho (com listas das existentes)

– usar desenhos de nodos da DAST para programação visual das novas regras

Será também prioritário criar mais *front-end*'s para linguagens usadas no ensino da programação, para que o sistema possa cumprir os seus objectivos pedagógicos.

Por último, e numa perspectiva de investigação será estudada a relação entre o sistema e as linguagens visuais de programação e a mais valia que poderá acrescentar. Uma linguagem visual é uma linguagem que usa uma notação predominantemente gráfica, isto é, um alfabeto formado por símbolos gráficos ou icónicos, sendo as frases construídas pela combinação espacial (a duas dimensões) desses símbolos (e não, apenas, pela sua concatenação). Existem muitos argumentos a favor das linguagens visuais. Esses argumentos baseiam-se essencialmente no facto de que os humanos processam mais rapidamente figuras do que texto. O raciocínio humano é bastante mais orientado à imagem. As pessoas adquirem informação a maior velocidade descobrindo relações gráficas em figuras complexas do que lendo texto. Daí também se defender, nesta tese, a importância da representação visual na animação de programas.

Muitas linguagens visuais baseiam-se no paradigma *rewriting*, ou seja, são processadas com base em regras de reescrita para descrever a transformação de cada figura; a solução constrói-se procurando fazer concordância de uma figura, ou parte dela, com a parte esquerda de uma regra, substituindo-a então, pela parte direita da mesma regra. Esta técnica é usada no sistema Alma para reescrever os nodos da DAST. Será interessante estudar até que ponto o sistema poderá ser usado para interpretar linguagens visuais.

7.3 Importância pedagógica do sistema Alma

Na medida em que o sistema Alma também está vocacionado para o ensino da programação, tem por objectivo ajudar, de alguma forma, os alunos a compreenderem os programas escritos por eles e pelo professor. A animação pode ser criada pelo próprio aluno ou então pelo professor se este quiser incluir a visualização dos programas no plano da aula.

As animações contribuem para o sucesso da aprendizagem visto ser mais fácil e intuitivo observar uma representação esquemática (visual) do que a linguagem textual usada para programar. Para o aluno tirar proveito da animação deve compreender o mapeamento entre o algoritmo e a sua representação gráfica. Os desenhos usados devem ser simples, quer na ideia que transmitem, quer graficamente. O aluno deve ainda poder *navegar* na animação quantas vezes entender, à velocidade desejada e em ambas as direcções. Com o sistema Alma o aluno pode visualizar o conteúdo das variáveis e a sua alteração sempre que uma instrução é executada. Esta facilidade vai permitir que o aluno compreenda o que faz cada operação, a

evolução dos valores das variáveis ao longo da execução e o objectivo final do programa.

Por fortes constrangimentos temporais não foi feito o estudo de impacto do sistema Alma no ensino da programação, como era desejável e como estava previsto à partida. Esta tarefa requer um estudo muito cuidado a nível de interfaces que terão que ser implementadas em conformidade. Relativamente às interfaces prevê-se a necessidade de desenvolvimento de vários níveis de interacção: no uso normal do sistema, uma interface que facilite a apreensão das matérias e o torne atractivo; em usos mais avançados, uma interface que permita com facilidade e segurança uma intervenção mais profunda no sistema de modo a realizar a preparação deste para o ambiente da aula.

Bibliografia

[AP02] Isabelle Attali and Didier Parigot. Smarttools project. http://www.dyade.fr/en/actions/smarttools/smarttools.html, 2002.

[Ber91] Yves Bertot. Occurences in debugger specifications. In *PLDI91*, 1991.

[BNR97] M. H. Brown, M. A. Najork, and R. Raisamo. A Java-based implementation of collaborative active textbooks. In *VL'97 - IEEE Symposium on Visual Languages*, pages 376–384. IEEE, September 1997.

[BS84] M. H. Brown and R. Sedgewick. A system for algorithm animation. In *SIGGRAPH'84*, volume 18, pages 177–186, Minneapolis, July 1984. ACM Computer Graphics.

[Bur01] Margaret Burnett. VL/HCC Bibliography. http://www.cs.orst.edu/~burnett/vpl.html, 2001.

[BV99] Peter Brummund and Ngozi V.Uti. Complete collection of algorithms animations. http://www.cs.hope.edu/~alganim/ccaa/index.html, 1999.

[CBC96] Paul Carlson, Magaret Burnett, and Jonathan Cadiz. A seamless integration of algorithm animation into a visual programming language. In *AVI'96 - International Workshop on Advanced Visual Interfaces*. acm, May 1996.

[Cre98] Rui Gustavo Crespo. *Processadores de Linguagens: da concepção à implementação*. IST Press, 1998.

[CSC94] CSC. Mathematical visualizations and animations. http://www.CSC.fi/math_topics/Movies/, 1994.

[Dav94] Dave. C algorithm animations in java. http://www.cm.cf.ac.uk/Dave/C/ANIM/anim.html, 1994.

[DF01a] Camil Demetrescu and Irene Finocchi. Leonardo. http://www.dis.uniroma1.it/~demetres/Leonardo, 2001.

[DF01b] Camil Demetrescu and Irene Finocchi. Smooth animation of algorithms in a declarative framework. *Journal of Visual Languages and Computing*, 12(3), 2001.

[Dui98] R. A. Duisberg. Animation using temporal constraints: An overview of the ANIMUS system. *Human-Computer Interaction*, 3(3):275–307, August 1998.

[Ell98] Conal Elliott. *Composing Reactive Animations*. Microsoft Research Graphics Group, 1998.

[GM00] Anabela Jesus Gomes and António José Nunes Mendes. Suporte à aprendizagem da programação com o ambiente SICAS. In *RIBIE 2000*, 2000.

[Goo99] Goodrich. Data structure and algorithm applets-interactive stack and queue adts. http://www.cgc.cs.jhu.edu/~goodrich/dsa/applets/index.html, 1999.

[Gra99] Calum. A. Mck. Grant. *Software Visualization in Prolog*. PhD thesis, Queen's College - Cambridge, 1999.

[Hau99a] Alejo Hausner. Bubblesort. http://www.cs.princeton.edu/~ah/alg_anim/gawain-4.0/BubbleSort.html, 1999.

[Hau99b] Alejo Hausner. Graham's scan. http://www.cs.princeton.edu/~ah/alg_anim/gawain-4.0/GrahamScan.html, 1999.

[HHR89] E. Helttula, A. Hyrskykari, and K. Raiha. Graphical specifications of algorithm animations with ALADDIN. In *22nd Hawaii International Conference on System Sciences*, January 1989.

[HPS+97] J. Haajanen, M. Pesonius, E. Sutien, T. Terasvirta, P. Vanninen, and J. Tarhio. Animation of user algorithms in the web. In *VL'97 - IEEE Symposium on Visual Languages*, pages 360–368. IEEE, September 1997.

[HVML02] P. Henriques, M. J. Varanda, M. Mernik, and M. Lenic. Automatic generation of language-based tools. In *LDTA - Workshop on Language, Descriptions, Tools and Applications (ETAPS'02)*, April 2002.

[Jar97] Duane J. Jarc. Interactive data structures visualizations. http://tangle.seas.gwu/~idsv/idsv.html, 1997.

[Jen96a] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-verlag, 2 edition, 1996.

[Jen96b] Kurt Jensen. Desenho de coloured petri nets para modelação e simulação de comportamento de sistemas. http://www.daimi.au.dk/designCPN, 1996.

[KA95] Hideki Koike and Manabu Aida. A bottom-up approach for visualizing program behavior. In *IEEE Workshop on Visual Languages*. IEEE, October 1995.

[KS98] Matthijs Kuiper and João Saraiva. Lrc—a generator for incremental language-oriented tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction*, volume 1383, pages 298–301. Springer-Verlag, April 1998.

[LF95] H. Lieberman and C. Fry. ZStep 95: A reversible, animated source code stepper. In *ACM Conference on Computers and Human Interface*, Denver, Colorado, April 1995.

[McW96] Jeffrey D. McWhinter. Algorithm explorer: A student centered algorithm animation system. In *VL'96 - IEEE Symposium on Visual Languages*, pages 174–181. IEEE, September 1996.

[Mic96] Amir Michail. Teaching binary tree algorithms through visual programming. In *VL'96 - IEEE Symposium on Visual Languages*, pages 38–45. IEEE, September 1996.

[MLAZ00] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Compiler/interpreter generator system LISA. In *IEEE Proceedings of 33rd Hawaii International Conference on System Sciences*, 2000.

[MM88] A. J. Mendes and Teresa Mendes. VIP - a tool to visualize programming examples. *Education and Application of Computer Technology*, 1988.

[Moh88] T. G. Moher. PROVIDE: A process visualization and debugging environment. In *IEEE Transactions on Software Engineering*, volume 14, pages 849–857, June 1988.

[MRRT99] Boris Melamed, Michael Rudolf, Olga Runge, and Gabriele Taentzer. The attributed graph grammar system - homepage. http://tfs.cs.tu-berlin.de/agg/, 1999.

[MS93] S. Mukherjea and J. T. Stasko. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *15th International Conference on Software Engineering*, pages 456–465, Baltimore, May 1993.

[Nor94] Norsoft. Magic animator - easy animation for windows - version 1.0. Programa e documentação, 1994.

[OPMS98] M. B. Ozcan, P. W. Parry, I. C. Morrey, and J. I. Siddiqi. Visualisation of executable formal specifications for user validation. In *Lecture Notes in Computer Science/Services and Visualization*, volume 1385, pages 142–157. Springer-verlag, 1998.

[RCWP92] G. C. Roman, K. Cox, C. Wilcox, and J. Plun. PAVANE: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(1):161–193, 1992.

[Rei85] Steven Reiss. PECAN: Program development systems that support multiple views. *IEEE Transactions on Software engineering*, 1985.

[Rei87] Steven Reiss. Working in the GARDEN environment for conceptual programming. *IEEE Software*, 1987.

[Rei90] Steven Reiss. Interacting with the FIELD environment. *Software Practice and Experience*, 1990.

[Rod96] Susan Rodger. The JAWAA homepage - java and web based algorithm animation. http://www.cs.duke.edu/csed/jawaa2/examples.html, 1996.

[San95] G. Sander. VCG - visualization of compiler graphs. Technical report, Universitat des Saarlandes, 1995.

[Sar99] João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Utrecht University, 1999.

[SBC96] John T. Stasko, D. Michael Byrne, and Richard Catrambone. Do algorithm animations aid learning? Technical Report GIT-GVU-96-18, Georgia Institute of Technology, Atlanta, August 1996.

[SDBP97] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization - Programming as a Multimedia Experience*. The MIT Press, 1997.

[Sta90] John T. Stasko. Simplifying algoritm animation with TANGO. In *IEEE Workshop on Visual Languages*. IEEE, October 1990.

[Sta96] John T. Stasko. Using student-built algorithm animations as learning aids. Technical Report GIT-GVU-96-19, Georgia Institute of Technology, Atlanta, August 1996.

[Sta99a] John Stasko. POLKA. http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html, 1999.

[Sta99b] John Stasko. SAMBA. http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html, 1999.

[Sta01] John Stasko. XTANGO. http://www.cc.gatech.edu/gvu/softviz/algoanim/xtango.html, 2001.

[VH99] M. J. Varanda and P. Henriques. Animação de algoritmos tornada sistemática. In *Workshop Computação Gráfica, Multimédia e Ensino*, February 1999.

[VH00] M. J. Varanda and P. Henriques. Visualização sistemática de programas. In *IV Simpósio Brasileiro de Linguagens de Programação*, May 2000.

[VH01] M. J. Varanda and P. Henriques. Visualization / animation of programs based on abstract representations and formal mappings. In IEEE, editor, *HCC'01 - 2001 IEEE Symposia on Human-Centric Computing Languages and Environments*, September 2001.

[VH03] M. J. Varanda and P. Henriques. Visualization / animation of programs in alma: obtaining different results. In *VMSE2003 - Symposium on Visual and Multimedia Software Engineering (HCC'03), New Zealand*, October 2003.

[Vod97] D. Vodislav. A visual programming model for user interface animation. In *VL'97 - IEEE Symposium on Visual Languages*, pages 348–357. IEEE, September 1997.

[WG84] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, 1984.

Apêndice A

Lista de funções usadas nas especificações do Alma

Algumas funções foram criadas para aceder aos nodos da DAST e respectiva informação:

```
setProdNumber: DASTNode X idProd ->DASTNode
getProdNumber: DASTNode -> idProd
getSymbol: DASTNode -> idSimb
getAttributes: DASTNode -> AlmaAtribs
setAttributes: DASTNode X AlmaAtribs -> DASTNode
setNodes: DASTNode X seq(DASTNode) -> DASTNode
getNodes: DASTNode -> seq(DASTNode)
size: DASTNode -> inteiro
equals: DASTNode X DASTNode -> Bool
```

Outras funções foram criadas para aceder a cada um dos atributos:

```
getName : DASTNode -> name
getType : DASTNode -> type
getValue: DASTNode -> value
setName : DASTNode X name -> DASTNode
setType : DASTNode X type -> DASTNode
setValue: DASTNode X value -> DASTNode
getTabId: DASTNode -> tabId
```

```
setTabId: DASTNode X tabId -> DASTNode
```

As funções para uso da tabela de identificadores serão:

```
createNode: idProd X idSimb X AlmaAtribs X seq(DASTNode) -> DASTNode
createTab: DASTNode -> tabId
putTableId: tabId X name X info -> tabId
putTableVal: tabId X name X value -> tabId
getTableVal: tabId X name -> value
getTableType: tabId X name -> type
getTableAddress: tabId X name -> address
```

Outras funções usadas nas regras de reescrita:

```
init: tabId X tabId X lista -> tabId
getNodeEnd: address -> DASTNode
getEndNode: DASTNode -> address
read:    -> value
write: value ->
exec: name X value X value -> value
sincronize: tabId X tabId -> tabId
```

Funções com tipos para regras de reescrita:

```
exec: name X value X type X value X type ->value
setValue: DASTNode X value X type X type -> DASTNode
write: value X type ->
```

Funções usadas nas regras de visualização:

```
drawRect: label X value ->
left_arrow:  ->
put: char ->
right_arrow: label ->
```

Apêndice B

Gramática estendida para construção de um *front-end*

A gramática de atributos estendida, em notação LISA, que especifica a linguagem usada nos exemplos apresentada na secção 5.4.5 surge agora completa:

```
import  "/Alma/BibAlma.lisa";

language notsosimp extends AlmaBase
{
   lexicon
         {
          Reserved  se | repetir | inicio | fim | ler | escrever
          Number     [0-9]+
          Identifier [a-z]+
          Operator   \+ | \- | \* | \/
          AssignOperator  \:\=
          Reloperator \> | \< | \<\= | >\= | \=\= | \!\=
          Separator  \[ | \] | \( | \)
          ignore     [\0x09\0x0A\0x0D\  ]+
         }


   rule Start
         {
          START ::= inicio STMTS fim compute
                  {
                     ALMA_ROOT<START,STMTS>
                  };
         }
```

```
  rule  Statements
        {
         STMTS ::= STMT STMTS compute
                 {
                    ALMA_STATS<STMTS, STMT,  STMTS[1]>
                 }
               | STMT  compute
                 {
                    ALMA_STATS_SING<STMTS, STMT>
                 };
        }
  rule  Statement
{
         STMT ::= ASSIGN  compute
                { ALMA_IDENT<STMT,ASSIGN>
                }
              | IF compute
                { ALMA_IDENT<STMT,IF>
                }
              | WHILE compute
                { ALMA_IDENT<STMT,WHILE>
                }
              | READ compute
                { ALMA_IDENT<STMT,READ>
                }
              | WRITE compute
                { ALMA_IDENT<STMT,WRITE>
                };
         }

  rule Assignment
        {
         ASSIGN ::= #Identifier \:\= EXPR compute
               {
                ALMA_ASSIGN_VAR<ASSIGN, #Identifier.value(), EXPR,"int">
               } ;
         }

  rule Expression
        {
         EXPR ::= EXPR \+ EXPR compute
 {
                 ALMA_OPER<EXPR[0],EXPR[1],EXPR[2], "+">
                }
```

```
              | EXPR \- EXPR compute
                {
                 ALMA_OPER<EXPR[0],EXPR[1],EXPR[2], "-">
                }
              | EXPR \* EXPR compute
                {
                 ALMA_OPER<EXPR[0],EXPR[1],EXPR[2],"*">
                }
              | EXPR \/ EXPR compute
                {
                 ALMA_OPER<EXPR[0],EXPR[1],EXPR[2],"/">
                };
        }


  rule Term1
        {
         EXPR ::= #Number compute
                {
                 ALMA_CONST<EXPR,#Number.value(),"int">
                };
        }
  rule Term2
{
         EXPR ::= #Identifier compute
                {
                 ALMA_VAR<EXPR,#Identifier.value(),"int">
                };
       }
  rule Conditional
       {
        IF ::= se CONDITION \[ STMTS \] \[ STMTS \]  compute
                { ALMA_IFELSE<IF,CONDITION,STMTS[0],STMTS[1]>
                }
           | se CONDITION \[ STMTS \]  compute
                { ALMA_IF<IF,CONDITION,STMTS>
                };
     }
  rule Cond
       {
        CONDITION ::= EXPR \> EXPR  compute
                { ALMA_OPERREL<CONDITION,EXPR[0],EXPR[1],">">
                }
              | EXPR \< EXPR  compute
                { ALMA_OPERREL<CONDITION,EXPR[0],EXPR[1],"<">
```

```
                }
              |  EXPR \<\= EXPR compute
                { ALMA_OPERREL<CONDITION,EXPR[0],EXPR[1],"<=">
                }
              | EXPR \>\= EXPR compute
                { ALMA_OPERREL<CONDITION,EXPR[0],EXPR[1],">=">
                }
              | EXPR \=\= EXPR compute
                { ALMA_OPERREL<CONDITION,EXPR[0],EXPR[1],"==">
                }
              | EXPR \!\= EXPR compute
                { ALMA_OPERREL<CONDITION,EXPR[0],EXPR[1],"!=">
                };
        }
   rule Whiledo
        {
         WHILE ::= repetir CONDITION \[ STMTS \]  compute
                { ALMA_WHILE<WHILE,CONDITION,STMTS>
                };
        }
   rule Read
        {
         READ ::= ler \( #Identifier \)  compute
                { ALMA_READ<READ,#Identifier.value(),"int">
                };
        }
   rule Write
        {
          WRITE ::= escrever \( #Identifier \)  compute
                { ALMA_WRITE<WRITE,#Identifier.value(),"int">
                };
        }

}
```

Apêndice C

Definição das funções para construção dos nodos da DAST

A gramática estendida mostrada no apêndice anterior utiliza, na definição das acções semânticas, funções pré-definidas pertencentes a um biblioteca do sistema Alma. Estas funções foram definidas usando o conceito de *template* do sistema LISA que permite especificar o cálculo de atributos envolvidos em cada uma dessas funções. O objectivo de cada função é construir um determinado tipo de nodo. O código desta biblioteca é apresentado a seguir:

```
template <attributes X_in, Y_in>
compute ALMA_ROOT
{
X_in.dast = new  Alma.CRoot(Y_in.tree);
}
template <attributes X_in, Y_in,  Z_in>
compute ALMA_STATS
{
X_in.tree = new  Alma.CStmtsNode(Y_in.tree,  Z_in.tree);
}
template <attributes X_in, Y_in>
compute ALMA_STATS_SING
{
        X_in.tree = new Alma.CStmtsNode(Y_in.tree);
}
template <attributes X_in, Y_in,  Z_in>
compute ALMA_ASSIGN
{
X_in.tree = new  Alma.CAssignNode(Y_in.tree,  Z_in.tree);
```

```
}
template <attributes X_in, Y_in,  Z_in, S_in>
compute ALMA_ASSIGN_VAR
{
X_in.tree = new  Alma.CAssignNode(new Alma.CVarNode(Y_in, S_in),  Z_in.tree);
}
template  <attributes  X_in, Y_in,S_in>
compute  ALMA_READ
{
X_in.tree = new  Alma.CReadNode(new Alma.CVarNode(Y_in,S_in));
}
template  <attributes  X_in, Y_in,S_in>
compute  ALMA_WRITE
{
X_in.tree = new  Alma.CWriteNode(new Alma.CVarNode(Y_in,S_in));
}
template  <attributes  X_in, Y_in, Z_in, S_in>
compute  ALMA_OPER
{
X_in.tree = new  Alma.COperNode(Y_in.tree,  Z_in.tree,  S_in);
}
template  <attributes  X_in, Y_in, Z_in, S_in>
compute  ALMA_OPERREL
{
X_in.tree = new  Alma.COperRelNode(Y_in.tree,  Z_in.tree,  S_in);
}
template  <attributes  X_in,  Y_in, Z_in>
compute  ALMA_VAR
{
X_in.tree = new  Alma.CVarNode(Y_in, Z_in);
}
template  <attributes  X_in,  Y_in, Z_in>
compute  ALMA_CONST
{
X_in.tree = new  Alma.CConstNode(Y_in, Z_in);
}
template  <attributes  X_in,  Y_in>
compute  ALMA_IDENT
{
X_in.tree = Y_in.tree;
}
template  <attributes  X_in,  Y_in, Z_in>
compute  ALMA_IF
{
X_in.tree = new  Alma.CIfNode(Y_in.tree, Z_in.tree);
```

```
}
template  <attributes  X_in, Y_in, Z_in, S_in>
compute  ALMA_IFELSE
{
X_in.tree = new  Alma.CIfElseNode(Y_in.tree,  Z_in.tree,  S_in.tree);
}
template  <attributes  X_in,  Y_in, Z_in>
compute  ALMA_WHILE
{
X_in.tree = new  Alma.CWhileNode(Y_in.tree, Z_in.tree);
}
language  AlmaBase  {

attributes  Alma.CAlmaNode *.tree;
attributes  Alma.CRoot  *.dast;
}
```

Apêndice D

Programa principal do sistema Alma

Neste apêndice estão incluídos o programa principal do sistema Alma (que é gerado pelo sistema LISA) e o código principal do *back-end*.
O programa principal do sistema Alma é gerado automaticamente aquando da construção do *front-end*, usando o sistema LISA. Este programa aplica o método de animação do *back-end* à árvore gerada pelo *front-end*.

```
//Lisa 2.0
//Laboratory for Computer Arhitectures and Programming Languages
import Lisa.Parser.*;
import Lisa.Scanner.*;
import Lisa.Interface.CMessage;
import Lisa.Interface.CTime;
import Lisa.Semantics.Evaluator.CEvaluatorDebugger;

import java.io.*;
public class Compile {
   public static void main(String[] args) throws Exception
     {
        new CMessage(System.out, System.out,System.err);
        CTime stopWatch = new CTime();
        System.out.println("Lisa Version 2.0");
        System.out.println("Generated compiler/interpreter");
        CScanner Scanner = new CScanner(new FileInputStream(args[0]), args[0]);
        CParser Parser = new CParser();
        System.out.println("Parsing");
        stopWatch.start();
        CSyntaxTree  Tree =Parser.parse(Scanner);
        System.out.println("File parsed in "+stopWatch.current()+" s.");
        System.out.println("vamos  escrever  a  arvore");
        if (Tree!= null)
          {
```

```
            System.out.println("Evaluating");
            CEvaluator Evaluator = new CEvaluator();
            stopWatch.start();
            Object o = Evaluator.evaluate(Tree.getRoot());
            double time = stopWatch.current();

            CSyntaxTree  s= (CSyntaxTree) o.getClass().getField("dast").get(o);
            Backend.animate(s);

            System.out.println(o);
            System.out.println("Program!!! evaluated in "+time+" s.");
          }

     }
}
```

O código principal respeitante à implementação do *back-end* do Alma é apresentado a seguir e divide-se claramente em método de reescrita e método de visualização, sendo a sua sincronização conseguida no método de construção da animação.

```
import  Lisa.Parser.CTreeNode;
import  Lisa.Parser.CSyntaxTree;
import  Lisa.Parser.CParseSymbolTerm;
import  Lisa.Parser.CParseSymbolNonTerm;
import  Lisa.Parser.CParseSymbol;
import  java.awt.Graphics;
import  java.util.Vector;

public  class  Backend{
        static  VRB  vis;
        static  RRB  rew;
        static TAB tab;
        static Vector reg;
        static Vector ss = new Vector();
        static boolean reescrevi=false;

        public  static  void  animate(CSyntaxTree btree){
            boolean  shownow;
            CTreeNode  n  =  btree.getRoot();
            printar(n);
            n=initTree(n,0);
            printar(n);
            reg= new Vector();
            vis=new  VRB(30);
            vis.createVRB();
            rew=new RRB(30);
```

```
    rew.createRRB();
    tab= new TAB();
    tab.createTAB(n);
    ss=visualize(n,ss);
    ss.addElement(new separatorShape());
    ss.addElement(null);
    //tocaadesenhar f= new tocaadesenhar(ss);
    //f.setVisible(true); /* 1ª visualização */
    n=rewrite(n);

    shownow=true;
    do {
           if (shownow) ss=visualize(n,ss);
           ss.addElement(new separatorShape());
           ss.addElement(null);
           reg.removeAllElements();
           n=rewrite(n);
    } while(!(reg.isEmpty()));
    tocaadesenhar f= new tocaadesenhar(ss);
    f.setVisible(true);
}


public  static  Vector  visualize(CTreeNode n,Vector ss){
        Vector  r  =new  Vector();
        boolean  found;
        if (n!= null && (!(((Alma.CAlmaNode)n).getName().equals("SKIP")))) {
            System.out.print("entrar  no  nodo");
            CTreeNode[]  nds=n.getNodes();
            if  (nds.length!=0)
                for(int i=0; i<nds.length;i++){
                      System.out.print("Vamos ver os filhos");
                      if ((nds.length>2) && (i==2))
                         if(((nds[1].getSymbol().getName()).equals("STATS")) &&
                            ((nds[2].getSymbol().getName()).equals("STATS"))){
                            ss.addElement(new separator2Shape());
                            ss.addElement(null);
                         }
                      ss=visualize(nds[i],ss);
                }
            else  System.out.print("Nao tem filhos");

            int  np=n.getProdNumber();
            r=vis.getRules(np);
            System.out.println("Temos " +  r.size() + " regras para " + np);
```

```
            found=false;
            int i=0;
            while(i<r.size()  &&  !found){
                    System.out.print("Tentar fazer o match");
                    found = ((visrule)(r.elementAt(i))).matching(n);
                    System.out.print("Encontrou=" + found);
                    i++;
            }
            if (i>0) i--;
            if (found) ss=((visrule)(r.elementAt(i))).draw(ss);

         }
        return ss;
}
public static CTreeNode rewrite(CTreeNode n){
      Vector  r  =new  Vector();
       boolean  found=false;
       reescrevi=false;

       if (n!= null && (!((Alma.CAlmaNode)n).getName().equals("SKIP")))) {

            int  np=n.getProdNumber();
            r=rew.getRules(np);
            System.out.println("Temos " +  r.size() + " regras para " + np);
            found=false;
            int i=0;
            while(i<r.size()  &&  !found){
                    System.out.println("Tentar fazer o match");
                    found = ((rewrule)(r.elementAt(i))).matching(n);
                    System.out.println("Encontrou=" + found);
                    i++;
            }
            if (i>0) i--;
            if (found) {n=((rewrule)(r.elementAt(i))).rewrite(n);
                        reescrevi=true;
                        reg.addElement(r.elementAt(i));
            }
            else {
               CTreeNode[]  nds=n.getNodes();
               if  (nds.length!=0){
                reescrevi=false;
                for(int p=0; ((p<nds.length) && (!reescrevi));p++){
                              System.out.println("Vamos reescrever os filhos");
                              rewrite(nds[p]);
                              }
```

```
                    if ((((Alma.CAlmaNode)n).getName().equals("REP"))&& (!reescrevi)){
                          resetNodes(n);
                          n=rewrite(n);
                    }
                   }
                   else  {System.out.println("Nao tem filhos");
                   }
               }

          }
          printar(n);
          return(n);
     }
}
```

Apêndice E

Exemplos de utilização do sistema Alma

Neste apêndice são apresentados dois exemplos de utilização do sistema Alma, onde são mostradas as visualizações geradas pelo sistema LISA (só para o primeiro exemplo) e as animações geradas pelo sistema Alma.

E.1 Primeiro exemplo

E.1.1 Código Fonte

O programa fonte a considerar no exemplo foi escrito numa linguagem algorítmica semelhante ao Pascal e tem o seguinte código:

```
inicio
a:=1
repetir a<8
   [a:=a+1
    se a==3 [a:=7]
   ]
se a==7 [escrever(a)]
fim
```

E.1.2 Visualizações geradas pelo LISA

As figuras E.1, E.2, E.3, E.4 e E.5 mostram alguns resultados visuais da análise da linguagem fonte produzidos automaticamente pelo gerador de compiladores LISA, usado, neste projecto, para construir os *front-end*'s do Alma. Concretamente vê-se o autómato reconhecedor dos símbolos terminais da linguagem; o diagrama de blocos que descreve a sintaxe da linguagem; uma parte de uma árvore de

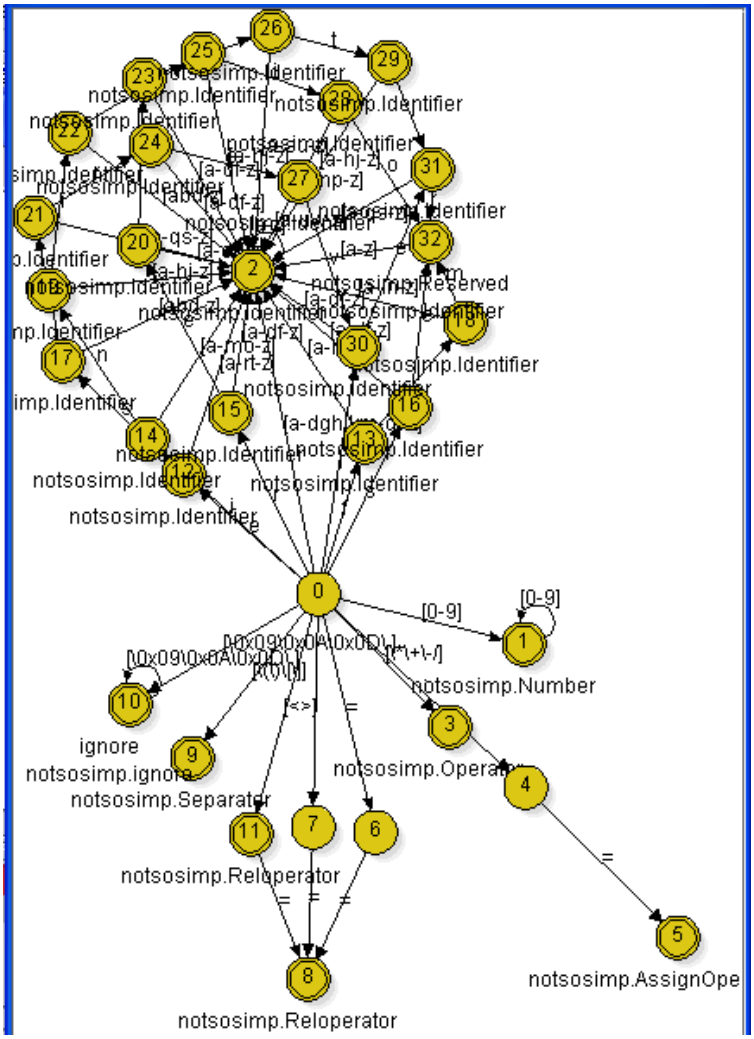

Figura E.1: Visualização do autómato da análise léxica

derivação (ou árvore de sintaxe) que corresponde à análise concreta de uma frase dessa linguagem (programa apresentado em E.1.1); uma visão estruturada desse código e, finalmente, a Árvore de Sintaxe Decorada (DAST) produzida pelo *front-end* gerado aquando do reconhecimento do referido programa fonte.

E.1.3 Animação gerada pelo Alma

As figuras E.6, E.7, E.8, E.9, E.10 e E.11 mostram seis momentos da animação produzida pelo *back-end* do Alma quando este processou o programa exemplo da secção E.1.1: a visualização da árvore de inicial; a visualização de alguns estados intermédios e a visualização final (após simular a execução completa do

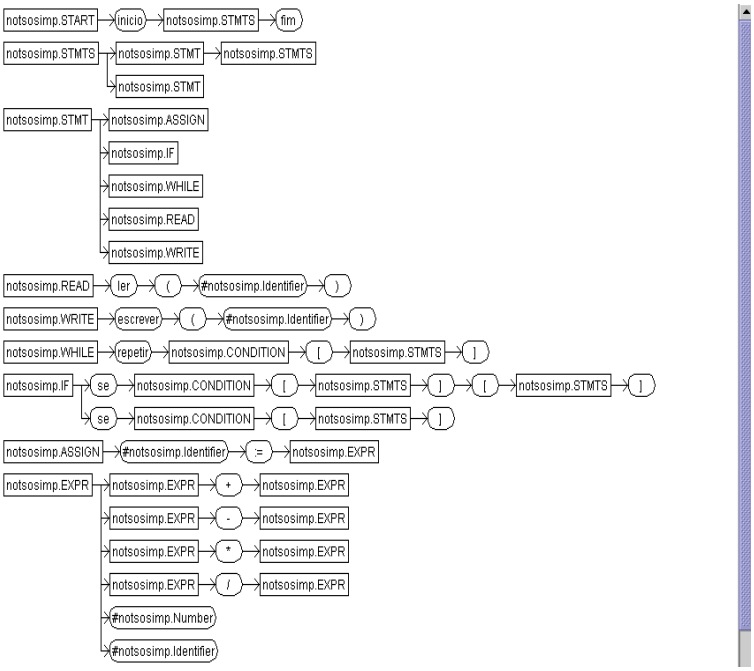

Figura E.2: Visualização de parte da gramática concreta

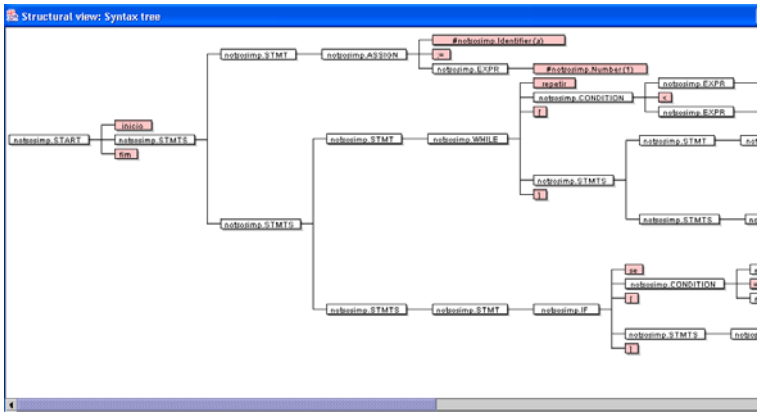

Figura E.3: Visualização de parte da árvore de sintaxe

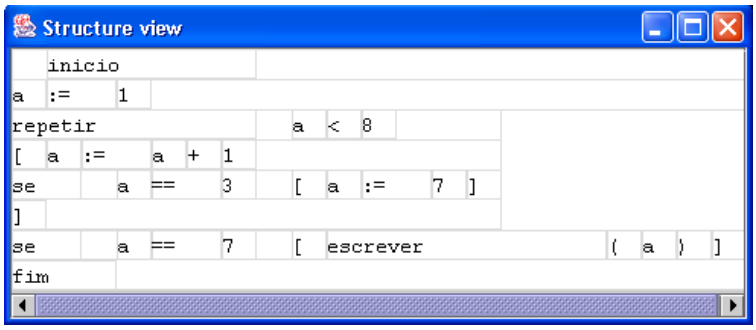

Figura E.4: Visão estruturada do programa fonte

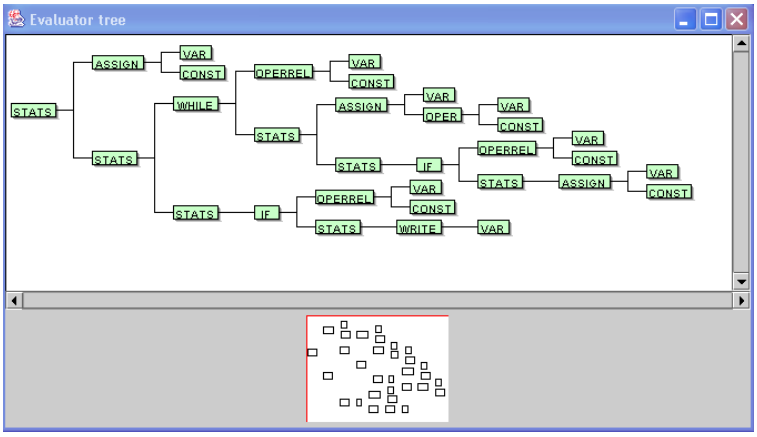

Figura E.5: Visualização da DAST gerada pelo *front-end* do Alma

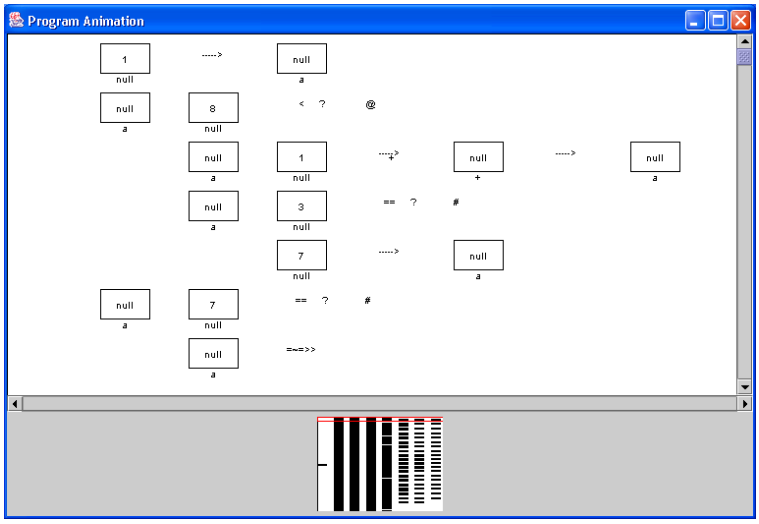

Figura E.6: Primeira visualização da animação do primeiro exemplo

programa — a árvore já não pode ser mais reescrita).

Os estados intermédios mostrados correspondem a pontos importantes do programa como por exemplo quando acaba uma iteração completa do ciclo ou quando uma condição se torna falsa fazendo desaparecer as instruções dependentes.

E.2 Segundo exemplo

E.2.1 Código Fonte

O programa fonte a considerar agora foi escrito na mesma linguagem algorítmica do exemplo anterior e tem o seguinte código:

```
inicio
a:=1
repetir a<=2
[a:=a+1
b:=a-1
repetir b<3 [b:=b+2 escrever(b)]
]
fim
```

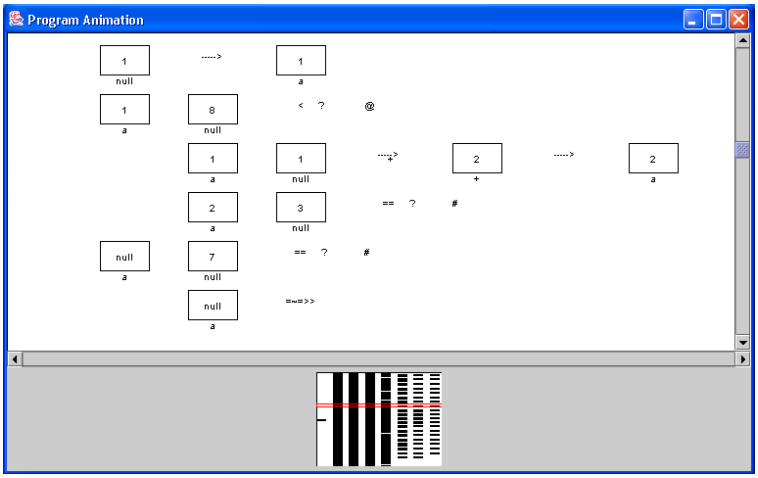

Figura E.7: Outra visualização da animação do primeiro exemplo

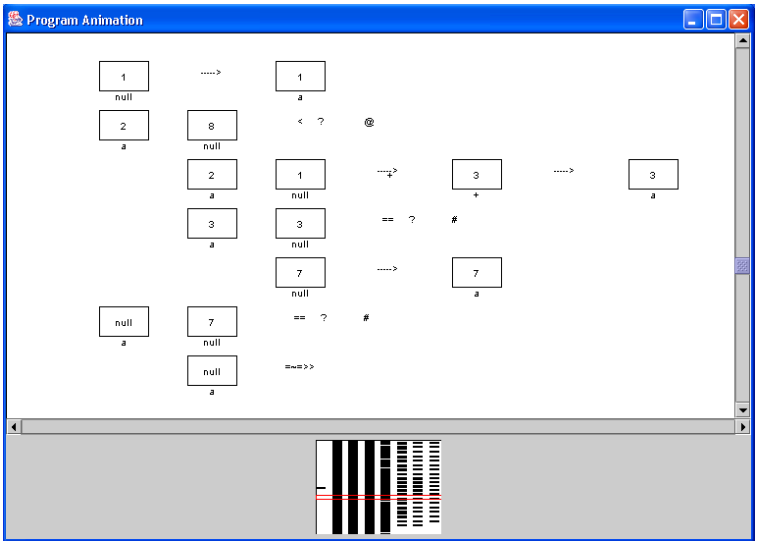

Figura E.8: Outra visualização da animação do primeiro exemplo

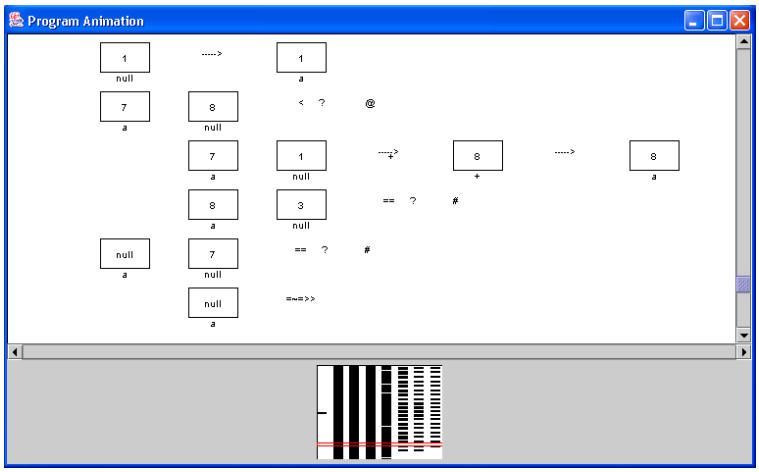

Figura E.9: Outra visualização da animação do primeiro exemplo

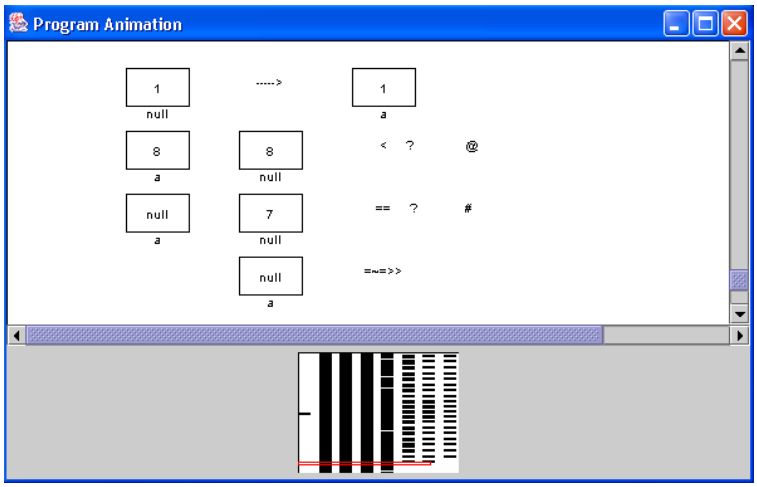

Figura E.10: Outra visualização da animação do primeiro exemplo

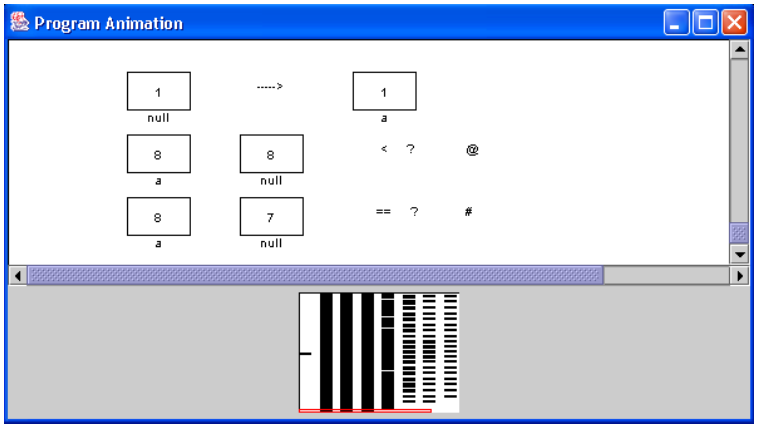

Figura E.11: Última visualização da animação do primeiro exemplo

E.2.2 Animação gerada pelo Alma

As figuras E.12, E.13, E.14, E.15, E.16 e E.17 mostram seis momentos da animação produzida pelo *back-end* do Alma quando este processou o programa exemplo da secção E.2.1: a visualização da árvore de inicial; as visualizações de estados intermédios e a visualização final.

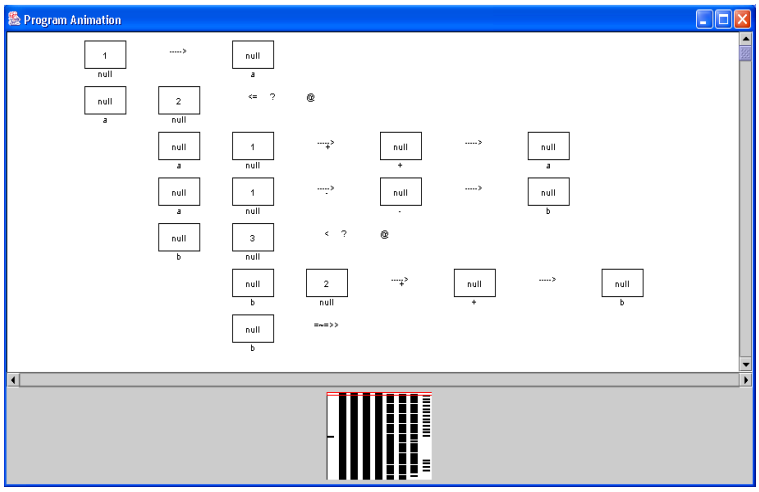

Figura E.12: Primeira visualização da animação do segundo exemplo

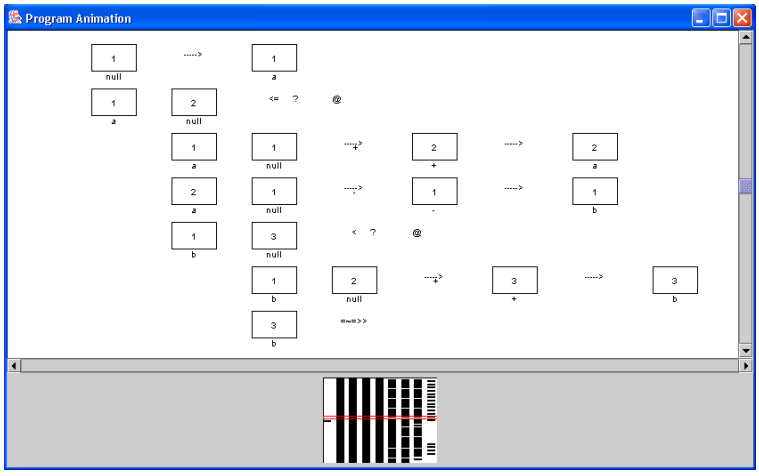

Figura E.13: Outra visualização da animação do segundo exemplo

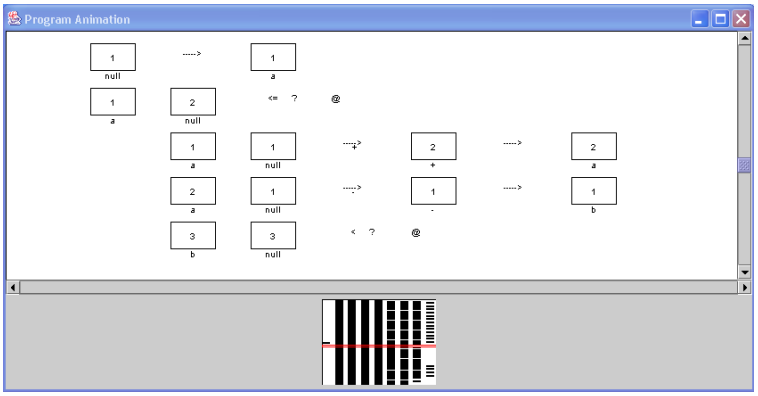

Figura E.14: Outra visualização da animação do segundo exemplo

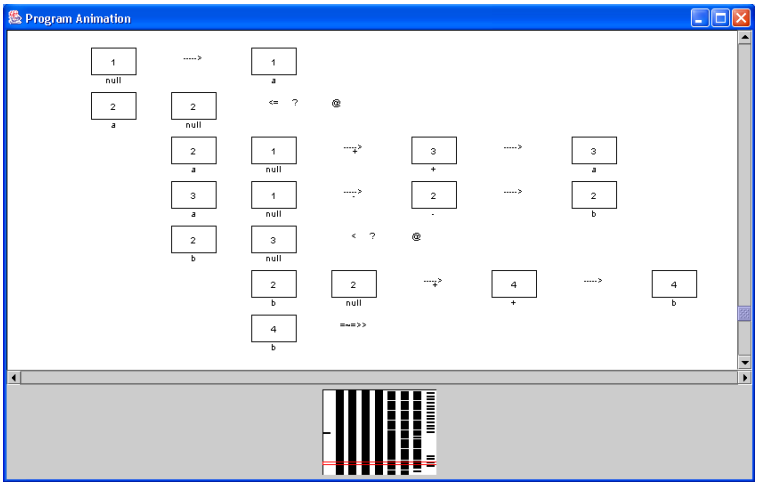

Figura E.15: Outra visualização da animação do segundo exemplo

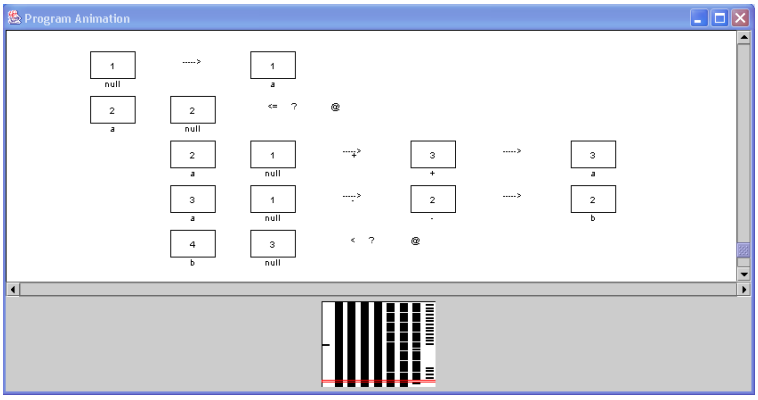

Figura E.16: Outra visualização da animação do segundo exemplo

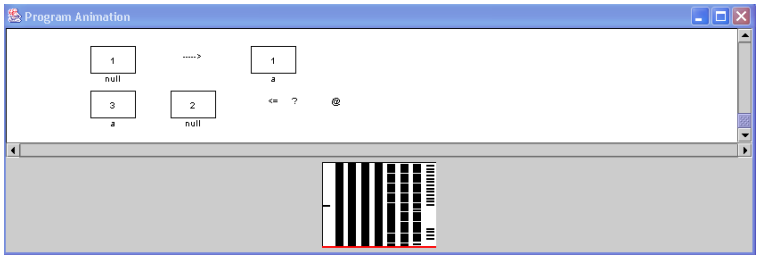

Figura E.17: Última visualização da animação do segundo exemplo

Apêndice F

Extensão da gramática do mini-prolog (construção da DAST)

Neste apêndice é mostrado um ficheiro de especificação LISA (gramática + acções semânticas), usado para construir o *front-end* de uma linguagem semelhante ao Prolog. O principal objectivo da construção deste *front-end* prende-se com a necessidade de provar que os elementos de uma linguagem pertencente ao paradigma lógico podem também ser mapeados nos nodos pré-definidos no *back-end* do Alma. Assim sendo, a representação interna do Alma pode ser suficientemente abrangente para que nada seja alterado no sistema quando se pretende animar programas pertencentes a outros paradigmas (é o caso do exemplo em causa).

Neste exemplo, a noção de regra ou facto (predicado) é mapeado no nodo `procdef` usado no paradigma imperativo para representar a definição de subprogramas. No fundo um predicado pode ser visto como um subprograma e a semântica associado é muito semelhante.

```
language SIMPLES {
lexicon {
  idmaius [A-Z]+
  idminus [a-z]+
  opad   \+ |\ -
  opmul  \* | \/
  par  \(  |  \)
  pontos  ; | \? | ! | =  | \, | \: | \ .
  WS  [\ \0x0D\0x0A\0x09]+
  ignore  #WS
}
```

```
attributes  CTreeNode *.tree;  CSyntaxTree PROG.dast; String  *.str;
rule AlmaAxioma{
     PROG ::= CLAUS  compute {PROG.dast = mkroot(CLAUS.tree);};
     }
rule CLS {
     CLAUS ::= CLAU  CLAUS  compute {CLAUS[0].tree = mkstats(CLAU.tree,CLAUS[1].tree);}
             | CLAU  compute {CLAUS.tree=CLAU.tree;};
     }
rule CL {
     CLAU  ::=  FACT  compute {CLAU.tree=FACT.tree;}
             |  RULE  compute {CLAU.tree=RULE.tree;};
     }
rule FCT {
     FACT  ::=  ATOMO  \.  compute {FACT.tree=mkprocdef(ATOMO.tree,null,ATOMO.nome);};
     }
rule ATM  {
     ATOMO  ::=  NOME #par LST #par compute {ATOMO.tree=LST.tree);
                                            ATOMO.nome=NOME.str;}
              |  NOME compute {ATOMO.tree=NULL;
                               ATOMO.nome=NOME.str;};
     }
rule LST  {
     LST  ::=  TERM \, LST  compute {LST[0].tree=mklst(TERM.tree,LST[1].tree);}
             |  TERM  compute {LST.tree=mklst(NULL,TERM.tree;};
     }
rule TER  {
     TERM  ::=  VAR  compute {TERM.tree=VAR.tree;}
              | CONST  compute {TERM.tree=CONST.tree;}
              | ATOMO compute {TERM.tree=ATOMO.tree;}
     }
rule RL  {
     RULE  ::=  ATOMO  \:  \-  LSTATOMOD \.  compute {RULE.tree=
                              mkprocdef(ATOMO.tree, LSTATOMOD.tree,ATOMO.nome);};
     }
rule  LSTATD  {
     LSTATOMOD ::= ATOMOD  \, LSTATOMOD  compute {LSTATOMOD[0].tree=
                                 mkstats(ATOMOD.tree,LSTATOMOD[1].tree);}
                 | ATOMOD  compute {LSTATOMOD.tree=ATOMOD.tree;};
      }
rule ATD {
     ATOMOD ::= ATOMO compute {ATOMOD.tree=ATOMO.tree;}
              | ATRIB compute {ATOMOD.tree=ATRIB.tree;}
              | RELOPER compute {ATOMOD.tree=RELOPER.tree;};
     }
rule ATRIBR {
```

```
     ATRIB ::= VAR is OPER compute {ATRIB.tree=mkassign(VAR.tree,OPER.tree);};
     }
rule OPERR {
     OPER ::= EXP '+' EXP compute {OPER.tree=mkoper(EXP.tree,EXP.tree,'+');}
            | EXP '-' EXP compute {OPER.tree=mkoper(EXP.tree,EXP.tree,'-');};
     }
rule RELOPERR {
     RELOPER ::= EXP '>' EXP compute {RELOPER.tree=mkoperrel(EXP.tree,EXP.tree,'>');}
               | EXP '<' EXP compute {RELOPER.tree=mkoperrel(EXP.tree,EXP.tree,'<');};
     }
rule EXPR {
     EXP ::= VAR  compute {EXP.tree=VAR.tree;}
           | CONST  compute {EXP.tree=CONST.tree;};
     }
rule VAR {
     VAR  ::=  #idmaius  compute { VAR.tree = mkvar(#idmaius.value());};
     }
rule CONST {
     CONST  ::=  #idminus  compute { CONST.tree  =  mkconst(#idminus.value());};
     }
rule NM  {
     NOME  ::=  #idminus  compute  {NOME.str=(String)#idminus.value();};
     }
}
```

Apêndice G

Exemplos de utilização do sistema AGG

Neste apêndice pretende-se mostrar mais dois exemplos de programas no sistema AGG, sistema de programação visual baseado em regras de reescrita de grafos que pertence ao tipo 0 e foi apresentado na secção 2.5.1. O primeiro estabelece regras para o casamento de dois seres humanos através da definição de atributos e relacionamento entre variáveis (figuras G.1 e G.2), e vem no seguimento do exemplo apresentado na figura 2.4. Em G.1 impõe-se que o valor do atributo `sexo`, nos nodos 1 e 2 do mesmo tipo `ser`, tem de ter um valor diferente; vendo-se em G.2 o resultado de aplicar essas regras a um cenário inicial (com 4 nodos do tipo `ser`, 2 femininos, 2 masculinos, 3 solteiros e 1 casado).

Na sequência do exemplo da figura 2.5, as figuras G.3 e G.4 representam uma situação de compra e venda onde se usa também a facilidade do sistema no estabelecimento de relações entre variáveis, para impor que o valor de um atributo de um nodo tenha de ser maior ou igual ao valor de um atributo de outro nodo (neste caso, de tipos diferentes). O cenário final que se vê na figura G.4, resulta de aplicar o programa da figura 2.5 (com a extensão da figura G.3) a um cenário inicial em que o cliente parte com 1500 escudos.

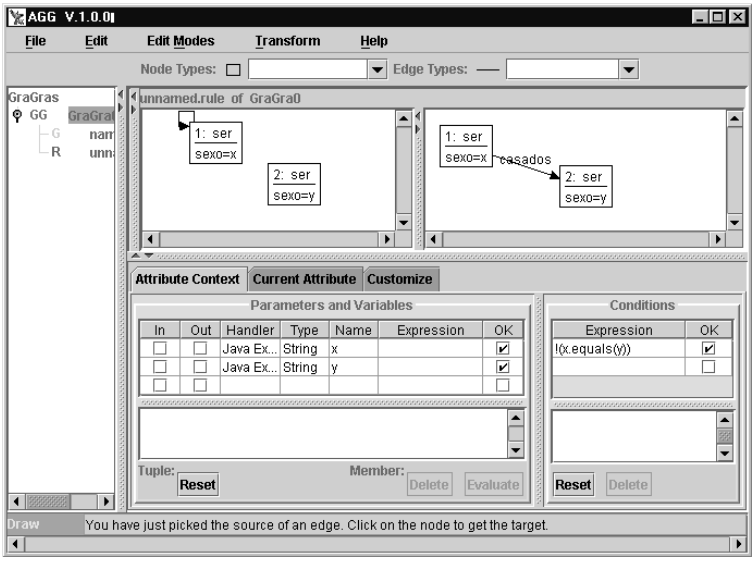

Figura G.1: Exemplo AGG com relacionamento de variáveis

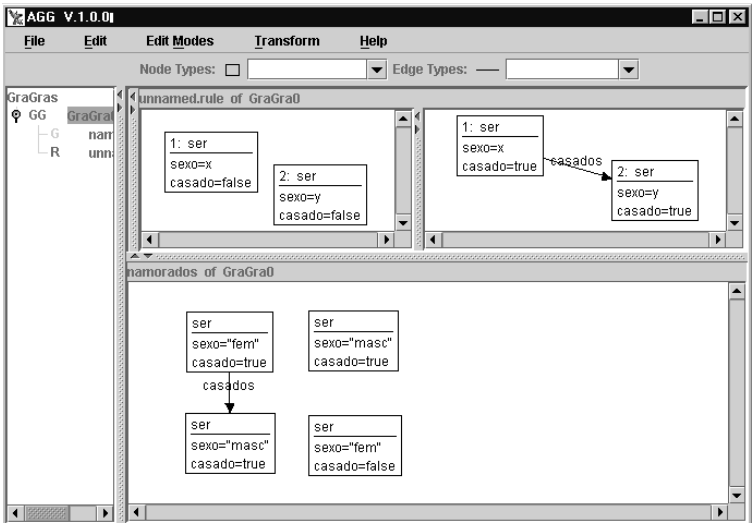

Figura G.2: Aplicação de regras no exemplo da figura G.1

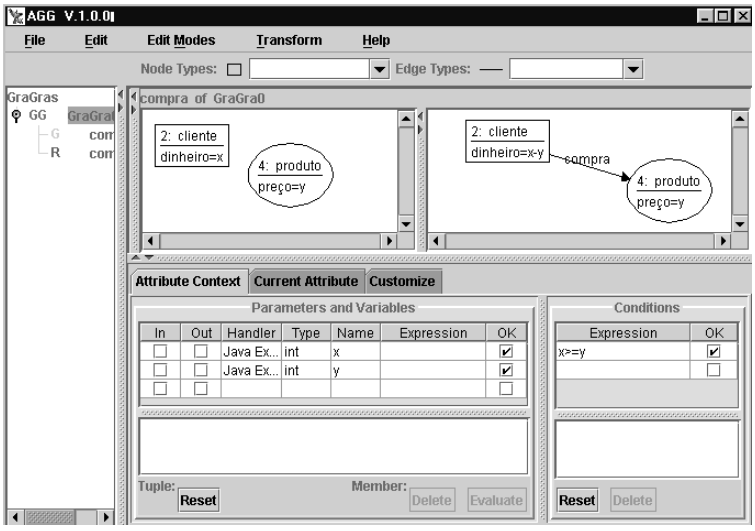

Figura G.3: Segundo exemplo AGG com relacionamento de variáveis

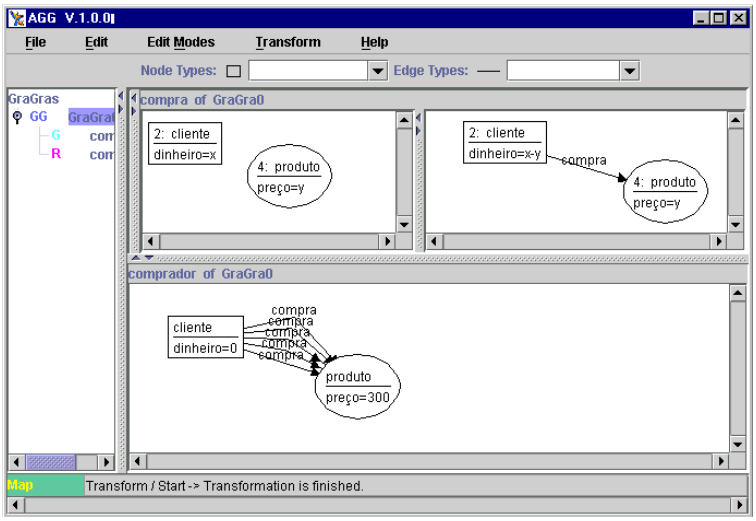

Figura G.4: Aplicação de regras no exemplo da figura G.3

Apêndice H

Código de animação de um algoritmo Depth First (JAWAA)

Neste apêndice é apresentado um exemplo de programa escrito na linguagem de especificação de animações do sistema JAWAA para criar a animação de um algoritmo de travessia Depth First.

```
#Depth First
Search Animation

#Begin Initialization
begin
node 1 120 30 20 black lightGray
node 2 70 120 20 black lightGray
node 3 170 120 20 black lightGray
node 4 120 200 20 black lightGray
node 5 220 200 20 black lightGray
node 6 170 250 20 black lightGray
node 7 20 200 20 black lightGray
node 8 70 250 20 black lightGray

connectNodes 9 1 3 black false
connectNodes 10 1 2 black false
connectNodes 11 3 4 black false
connectNodes 12 3 5 black false
connectNodes 13 4 5 black false
connectNodes 14 4 2 black false
connectNodes 15 4 8 black false
connectNodes 16 7 2 black false
connectNodes 17 4 6 black false
connectNodes 18 6 5 black false
connectNodes 19 8 7 black false
```

```
text pre 250 30 "Preorder: " black
text post 250 60 "Postorder: " black
stack s 300 150 0 black red
text ts 260 150 "Stack:" black
end

#End Initialization

#Begin Animation
marker 20 1 10 black green
push s 1
changeParam 1 bkgrd green
text t1 315 30 "1" green
moveMarker 20 1 2 10 black green
text t2 330 30 "2" green
push s 2
changeParam 2 bkgrd blue
pop s
text t3 315 60 "2" blue
moveMarker 20 2 1 10 black blue
moveMarker 20 1 3 9 black green
changeParam 3 bkgrd green
text t4 345 30 "3" green
push s 3
moveMarker 20 3 4 11 black green
changeParam 4 bkgrd green
text t5 360 30 "4" green
push s 4
moveMarker 20 4 8 15 black green
changeParam 8 bkgrd green
text t6 375 30 "8" green
push s 8
moveMarker 20 8 7 19 black green
text t7 390 30 "7" green
push s 7
changeParam 7 bkgrd blue
pop s
text t20 330 60 "7" blue
moveMarker 20 7 8 19 black blue
changeParam 8 bkgrd blue
pop s
text t8 345 60 "8" blue
moveMarker 20 8 4 15 black blue
moveMarker 20 4 6 17 black green
```

```
changeParam 6 bkgrd green
push s 6
text t9 405 30 "6" green
moveMarker 20 6 5 18 black green
text t15 420 30 "5" green
push s 5
changeParam 5 bkgrd blue
pop s
text t10 360 60 "5" blue
moveMarker 20 5 6 18 black blue
changeParam 6 bkgrd blue
pop s
text t11 375 60 "6" blue
moveMarker 20 6 4 17 black blue
changeParam 4 bkgrd blue
pop s
text t12 390 60 "4" blue
moveMarker 20 4 3 11 black blue
changeParam 3 bkgrd blue
pop s
text t13 405 60 "3" blue
moveMarker 20 3 1 9 black blue
changeParam 1 bkgrd blue
pop s
text t14 420 60 "1" blue
delete 20
```